# How Model Growth, Boundary Operators, and Recursion Improve Scaling Laws

**Anonymous authors**
Paper under double-blind review

## Abstract

Scaling laws predict how loss decreases as computation increases. Yet how architecture affects scaling laws is rarely examined. While a better constant brings scale-invariant compute-efficiency gains, a better exponent leads to power-law improvements in performance as computation increases. In this work, we comprehensively study how model growth, recursion, and boundary operators affect scaling coefficients. Specifically, we run more than 30 distinct scaling ladders over compute budgets of approximately $10^{18}$ to $10^{20}$ FLOPs. We find that better architectures can improve the *exponents*. In particular, our best architectural variant, *Untied-Grow*, achieves a $1.55\times$ compute-efficiency gain at $10^{20}$ FLOPs, and the trend extrapolates to $10^{21}$ FLOPs, where our model matches GPT-3 13B on CORE. We find that weight sharing is suboptimal in fresh-token training. However, recursion has a useful regularizing effect in data-constrained, multi-epoch training. In this regime, we find that it is compute-optimal to increase the number of loops with scale, outperforming model-size scaling even with tuned weight decay. To understand these gains, we examine computational depth and find that exponent improvements correlate with faster growth in computational depth.

## 1 Introduction

*Compute-optimal scaling laws* predict loss as a function of computational budget, specifying how to allocate that budget between training data and model size (Kaplan et al., 2020; Hoffmann et al., 2022). These laws follow a power law governed by scaling constants and exponents. The constants govern the vertical translation of loss curves, while the exponents determine how quickly loss decreases with compute (Kaplan et al., 2020; Hoffmann et al., 2022). Modifying even the constants can have a significant effect on common practice. For example, Qiu et al. (2026) and Liu et al. (2025) recently showed that, with the correct hyperparameter scaling, the Muon optimizer can provide a 40% compute-efficiency gain over AdamW across scales. Muon is thus a promising candidate for the default optimizer, challenging Adam after nearly a decade of dominance. This success motivates innovations in deep learning that challenge long-standing assumptions.

In this paper, we focus on neural network architecture and explore how architectural interventions alter scaling constants and exponents. Conventional wisdom holds that changes to the architecture generally affect only the scaling constants or have a small effect on the overall scaling law (Bansal et al., 2022; Hestness et al., 2017; Chen et al., 2026; Patel & Han, 2026). An improvement in the exponent, however, would yield compute-efficiency gains that increase with scale, making even a small change consequential at larger budgets.

To explore how architectural interventions can alter scaling laws, we focus on the idea of *computational depth*. Greater depth enables efficient hierarchical representations at increasing levels of abstraction (LeCun et al., 2015; Telgarsky, 2016) and can unlock new capabilities (Wang et al., 2026b; Merrill & Sabharwal, 2026). Nevertheless, naively increasing depth is not compute-optimal, and its benefits can taper off quickly (Petty et al., 2024; Sun et al., 2025). To address these challenges, we consider *model growth*, whereby we increase the depth of a model during training, saving compute early while enabling greater depth later. We also consider *looping*, or *recursive depth*, which applies the same core block of layers several times within a forward pass (Geiping et al., 2025).



Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.

We summarize some of our key results in Figure 1. We express efficiency gains as *compute multipliers*: the ratio of the compute a standard transformer needs to the compute a variant needs to reach the same loss. An exponent improvement produces a multiplier that increases with scale. We find:

- Model growth improves the compute-optimal scaling exponent with either tied or untied weights, yielding efficiency gains that increase with scale. Tied growth retains this improvement without adding parameters, reaching a $1.36\times$ compute multiplier at $10^{20}$ FLOPs.
- The exponent improvement in fixed-loop transformers comes from the *boundary operator*, rather than weight sharing itself. Adding the operator to a standard transformer, without looping, yields a $1.25\times$ compute multiplier at $10^{20}$ FLOPs.
- In fresh-token, compute-optimal training, independent weights outperform shared weights in our matched comparisons. Untying improves the scaling constant while leaving the exponent approximately unchanged.
- Our best architecture, Untied-Grow, combines untied model growth with the boundary operator, reaching a $1.55\times$ compute multiplier at $10^{20}$ FLOPs. A 7.4B model trained at $8\times$ the largest fitted compute follows the extrapolated loss curve (Figure 3).
- In the data-constrained regime, weight sharing instead provides useful regularization. When training on 100M unique tokens for ten epochs, we find that the optimal loop count increases with compute. Scaling the loop count at fixed weight decay matches the best loss from model-size scaling with tuned weight decay using $2.2\times$ less compute.
- Model growth and looping enable greater usable computational depth at a fixed training budget. Measurements of KL effective depth support this interpretation, connecting the scaling gains to the number of layers that meaningfully influence the predictive distribution (Section 3.4).

These results show that architectural changes can improve scaling exponents, yielding gains that grow with compute. They also clarify when to reuse weights: untied growth is more compute-efficient on fresh data, while looping becomes advantageous when repeated data makes overfitting the limiting factor.

## 2 PRELIMINARIES

### 2.1 SCALING LAWS

A scaling law predicts losses as a sum of power laws. Hoffmann et al. (2022) write loss as three terms: a power law in the parameter count $N$, a power law in the number of training tokens $T$, and an irreducible floor $E$, giving $L(N,T) = E + A(N/N_0)^{-\alpha} + B(T/T_0)^{-\beta}$, where $N_0$ and $T_0$ are reference parameter and token counts, and $A$, $B$, $\alpha$, and $\beta$ are fitted empirically.

**Compute-optimal Scaling.** Choosing the best allocation between training data and model parameters at every budget gives the compute-optimal scaling law,

$$L(C) = E + A\left(\frac{C}{C_0}\right)^{-\gamma}, \tag{1}$$

where $A$ and $\gamma$ are fitted and $C_0$ is a reference compute budget. Each architecture thus defines a family of models, or a scaling ladder. Scaling laws also depend on the compute-optimal recipe. We will revisit the effect of recipes in Section 3.

**Compute Multipliers.** To compare two scaling laws, we measure the compute required to reach the same loss under each law. Compute multipliers are the ratios of these compute requirements. To reduce fitting error, we use the loss of the standard transformer as the target loss and estimate the compute required by other architecture variants through linear interpolation in log-loss and log-compute. In the figures, the x-axis shows the compute required by the baseline to reach the target loss. The inset arrow in the bottom-right panel of Figure 1 provides an illustration.

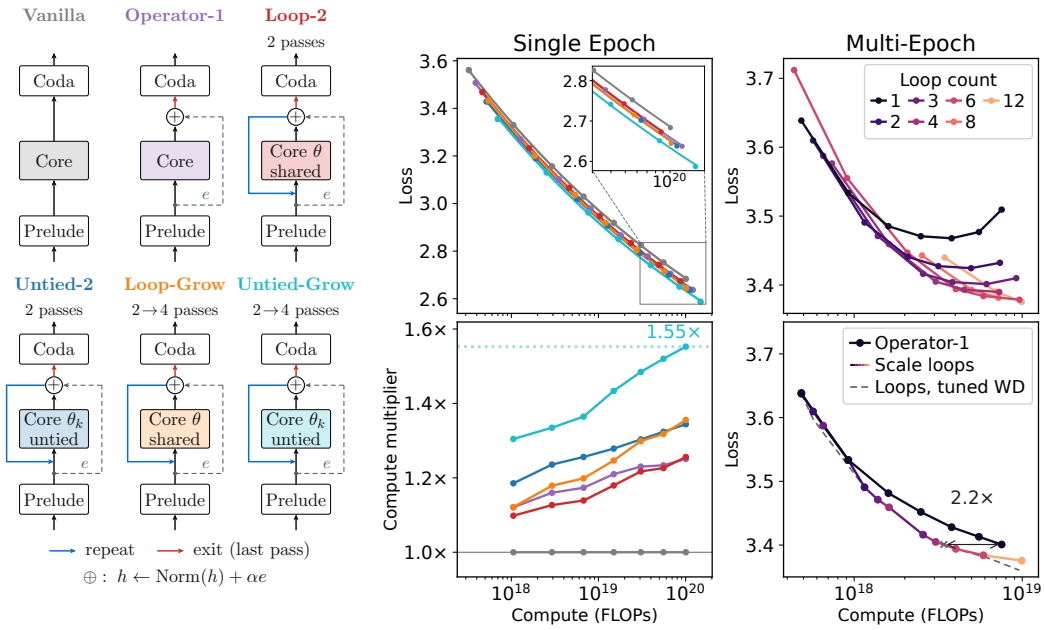



Figure 1: **Model growth, recursion, and boundary operators improve scaling in single epoch and multi-epoch training. Left:** illustrations for the major architectures. **Middle:** fresh-token loss and compute multipliers. Untied-Grow reaches $1.55\times$ at $10^{20}$ FLOPs. **Right:** training on 100M tokens for ten epochs. Scaling loops matches the best weight-decay-tuned model-size scaling with $2.2\times$ less compute.

### 2.2 Architectures

All architectures are based on a pre-norm dense transformer with a fixed width-to-depth ratio of 128. Every model we train can be expressed as

$$h_0 = e = \mathcal{P}(s), \quad z_k = \mathcal{R}_{\theta_k}(h_{k-1}), \quad h_k = \phi(z_k, e) \ \ (k = 1, \ldots, K), \quad y = \mathcal{C}(h_K). \tag{2}$$

Here, $\mathcal{P}$, $\mathcal{R}$, and $\mathcal{C}$ are the consecutive stages of transformer blocks: prelude, core, and coda (Geiping et al., 2025). The operator $\phi$ mixes the state with the prelude output $e$ between core passes and before the coda. $\theta_k$ are the weights of the $k$th pass, and $K$ is the loop count. We next explain how all architectures fit into this formulation. To start, we recover standard transformers (*Vanilla*) by setting $K = 1, \phi(h, e) = h$. Except for the shape-controlled Vanilla variants, all architectures are shown in Figure 1.

**Boundary operator (BO).** In looped transformers, the mixing operator $\phi$ is typically not the identity in its state argument. Instead, we have *boundary operators* (Geiping et al., 2025; Prairie et al., 2026):

$$\mathrm{BO}(h, e) = \mathrm{Norm}(h) + \alpha\, e, \tag{3}$$

where Norm is a normalization layer and $\alpha$ is a hyperparameter. Starting from Vanilla and setting $\phi$ to BO gives a new architecture, *Operator-1*. Both normalization and prelude injection are essential to the performance (Table 5 and Figure 14a).

**Recursion.** Looped transformers re-apply the core blocks for multiple passes. In our formulation, we set $K$ to be larger than one for Operator-1 and tie the weights, $\theta_1 = \cdots = \theta_K$. Note that the core blocks are applied $K$ times, so the model stores $P + C + D$ blocks and executes $P + KC + D$. We name these architectures *Loop-K*; *Loop-2* is the specific case with two recurrences.

**Untying.** Starting from *Loop-K*, we relax the tying condition $\theta_1 = \cdots = \theta_K$ and call the resulting architectures *Untied-K*. Thus, *Untied-2* is the same as Loop-2 except that it has more free parameters. Note that Loop-1 and Untied-1 are simply Operator-1.

**Model growth.** Using model growth, we start training at a small loop count $K$ and raise it partway through. In the main text, we specifically go from two passes to four passes and defer results for other growth schedules to Appendix A.2.3. Therefore, we have *Loop-Grow* for the tied case (McLeish et al., 2025) and *Untied-Grow* for the untied case (Du et al., 2024). The full training procedure for Untied-Grow is given in Algorithm 1.

**Shape Control.** Because setting $K$ larger than 1 changes the width-to-depth ratio, we provide two ablation architectures, *Deep Vanilla* and *Deep Vanilla Grow*, that match Untied-2 and Untied-Grow but omit the boundary operator. As we will show later in Section 3, these two ablations isolate the effects of aspect ratio and model growth from those of the boundary operator.

# 3 Compute-Optimal Scaling on Fresh Tokens

In this section we compare the compute-optimal scaling laws of the model families of Section 2.2 on fresh tokens, paying special attention to whether the gaps between them widen with scale or stay constant. Throughout, we train on FineWeb (Penedo et al., 2024) with the GPT-2 tokenizer (Radford et al., 2019). We describe the main setup here and defer details to Appendix A.

## 3.1 Compute-Optimal Recipe

While compute-optimal scaling laws prescribe the allocation of data and parameters (Hoffmann et al., 2022), training with larger budgets without additional adjustments can lead to training instability and suboptimal scaling behavior (Rae et al., 2021; Qiu et al., 2026). Hence, achieving optimal loss across compute budgets also requires prescriptive hyperparameter scaling (Yang et al., 2022; Qiu et al., 2026). We further find that architecture-specific hyperparameters can substantially affect the loss achieved at larger compute budgets. In this section, we explain the architecture-specific scaling choices and the scaling recipes we use. We provide additional details in Appendix A.3.

**Architecture-specific Choices.** We make three choices across the model families (Figure 6): **(i)** We scale the prelude, core, and coda evenly as we increase depth. Larger or smaller core allocations fall off the compute-optimal frontier. **(ii)** The optimal loop count is between one and two for fresh tokens, so we experiment only with Loop-2 and Untied-2. This differs in the data-constrained regime (Section 4). **(iii)** Starting from two passes, we find that four is the best target pass count for growth variants, so we experiment only with Loop-Grow and Untied-Grow.

**Scaling Recipes.** Before running the final scaling ladder, we use four sweeping and fitting stages: **(i) base hyperparameter tuning.** We perform two rounds of sequential 1D hyperparameter sweeps (Wen et al., 2026), tuning each parameter individually to approximate global optimality. **(ii) tokens per parameter (TPP) fitting.** We vary the model size at five compute budgets and find the optimal TPP (Hoffmann et al., 2022). **(iii) transition point fitting.** For growth families, we sweep the transition point for model growth at three compute budgets and take the average as the optimal transition point. **(iv) learning rate fitting.** We fit an empirical power-law rule for learning rates as a function of parameter count using three compute budgets. Scaling additional hyperparameters doesn't further improve the scaling behavior (Appendix A.3.4).

## 3.2 Compute-Optimal Scaling Results

Transferring compute-optimal recipes across architectures inhibits the full potential (Table 7 and Figure 15b). To ensure a fair comparison, we use an independently fitted compute-optimal recipe for each architecture. To compare the scaling constants and exponents of different architectures, we first fit the irreducible loss $E$ in Equation 1 using a Huber loss for Vanilla (Hoffmann et al., 2022). Then, for the remaining architectures, we fit an affine relationship in log–log space, $\log(L - E) = -\gamma \log(C/C_0) + \log A$, to obtain the exponent $\gamma$ and the constant shift $\log A$. $C_0$ is set to the compute

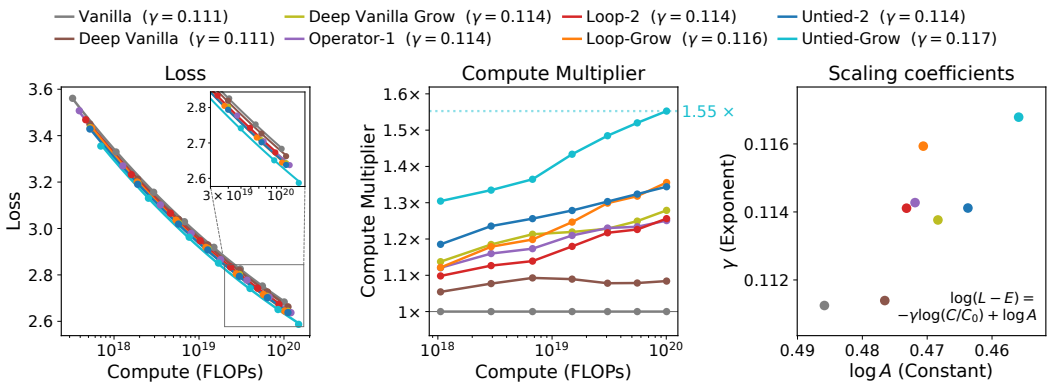



Figure 2: **Model growth and the boundary operator improve exponents and constants, and untying improves constants. Left:** validation loss against training compute for eight compute-optimal ladders. **Middle:** compute multipliers for Vanilla to achieve the same loss. **Right:** fitted exponent $\gamma$ against constant $\log A$ for each arm, where $\gamma$ controls the slope and $\log A$ controls the vertical translation of scaling curves; better is up and to the right. A better constant corresponds to a scale-invariant compute multiplier, and a better exponent corresponds to a growing one. The standard errors for $\gamma$ and $\log A$ are smaller than $0.001$. The legend shows each arm's fitted exponent. Figure 14b shows the same ladders against wall-clock time.

Table 1: **Changes in scaling constants and exponents.** We group architecture pairs that experience the same architectural intervention. We report the differences in exponents and constants. Positive $\Delta\gamma$ and negative $\Delta \log A$ indicate improvements. Standard errors are smaller than $0.001$ for both coefficients. A diagram of these coefficients is shown in the right panel of Figure 2.

| Intervention | Comparison | $\Delta\gamma$ ($\uparrow$) | $\Delta \log A$ ($\downarrow$) |
|---|---|---|---|
| Boundary operator | Vanilla $\rightarrow$ Operator-1 | $+0.003$ | $-0.014$ |
| | Deep Vanilla $\rightarrow$ Untied-2 | $+0.003$ | $-0.013$ |
| Recursion | Operator-1 $\rightarrow$ Loop-2 | $0.000$ | $+0.001$ |
| | Operator-1 $\rightarrow$ Untied-2 | $0.000$ | $-0.008$ |
| Untying | Loop-2 $\rightarrow$ Untied-2 | $0.000$ | $-0.009$ |
| | Loop-Grow $\rightarrow$ Untied-Grow | $+0.001$ | $-0.015$ |
| Growth | Loop-2 $\rightarrow$ Loop-Grow | $+0.002$ | $-0.003$ |
| | Untied-2 $\rightarrow$ Untied-Grow | $+0.003$ | $-0.008$ |
| | Deep Vanilla $\rightarrow$ Deep Vanilla Grow | $+0.002$ | $-0.008$ |
| Shape | Vanilla $\rightarrow$ Deep Vanilla | $0.000$ | $-0.009$ |

budget used to tune Vanilla, $1.06 \times 10^{18}$ FLOPs. With the fitted exponents and constants, we can compare the improvements from different architectural interventions: exponent improvements ($\gamma \uparrow$) lead to compute multipliers that grow with scale; constant improvements ($\log A \downarrow$) lead to scale-invariant compute multipliers. Although fitted scaling law coefficients depend on the specific fitting choices we make here, compute multipliers are estimated only by linear interpolation. In this section, we draw conclusions from scaling coefficients, but the same conclusions can be drawn from looking at compute multipliers. Figure 2 shows the scaling ladders, compute multipliers, and fitted scaling coefficients, and Table 1 shows the coefficient changes for different architectural interventions.

**Main results.** From Table 1 and Figure 2, we observe: **(i)** Boundary operators improve both exponents and constants. The gains are similar for different starting transformer shapes. **(ii)** Untied growth improves both exponents and constants. The gains are similar with and without boundary operators. **(iii)** While untying has a constant shift of $-0.009$ for models without growth, untying the growth models improves the exponent by $0.001$ and shifts the constant by $-0.015$. In addition, Loop-2 benefits less from growth than Untied-2. These two observations suggest that growth and weight tying interact. Growing with tied weights is less efficient than growing with untied weights.

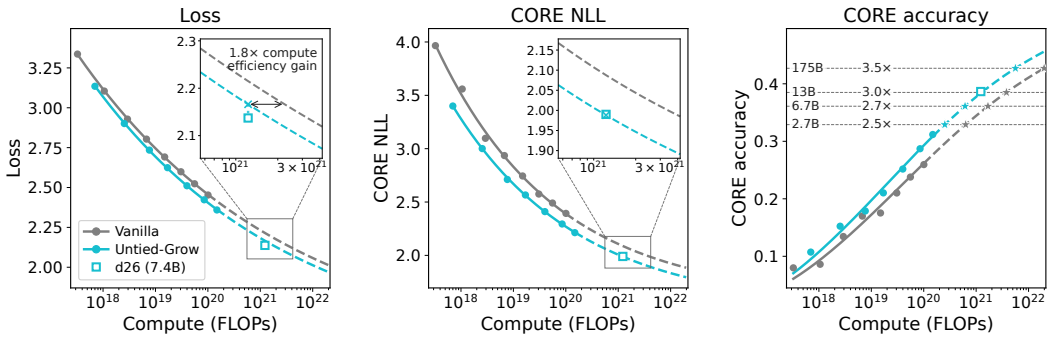



Figure 3: **The scaling law fitted on $10^{18}$–$10^{20}$ FLOPs predicts an Untied-Grow run at $8\times$ the fitted compute, and the compute-efficiency advantage over Vanilla widens with scale.** Ladders are trained on FineWeb-Edu, solid lines are fits, dashed lines are their extrapolation, and the square is a 7.4B Untied-Grow run not used in any fit. **Left:** validation loss on FineWeb-Edu, with a shared irreducible loss fitted on Vanilla. The run lands $0.03$ below the forecast. The inset arrow marks the compute multiplier at the run's budget: Untied-Grow reaches Vanilla's loss with $1.8\times$ less compute. **Middle:** answer NLL, where the run lands within $0.002$ of the forecast. **Right:** CORE accuracy predicted from compute (Appendix C.2), where the run reaches $0.387$ against a forecast of $0.384$. Horizontal lines are GPT-3 CORE scores from 2.7B to 175B, and stars mark where each curve crosses them. The Vanilla-to-Untied-Grow compute ratio at those crossings grows from $2.5\times$ to $3.5\times$, so the exponent improvement of Figure 2 carries over downstream.

**(iv)** Tied recursion does not improve scaling laws. On the other hand, untied recursion improves the constant by $-0.008$, similar to the constant improvement from Vanilla to Deep Vanilla. Motivated by the constant improvement with a deeper model, we run scaling ladders for even deeper Vanilla models, which turn out to be worse than Deep Vanilla (Figure 15c).

We find that boundary operators and model growth improve constants and exponents. In the fresh-token regime, untying weights always works better, especially for growth. This makes sense intuitively because weight tying limits the flexibility of the model and the extra flexibility provided by growth is also limited by weight tying. As a result, our best-performing model is *Untied-Grow*, which achieves a $1.55\times$ compute-efficiency gain at $10^{20}$ FLOPs. A natural question to ask next is whether the scaling law extrapolates.

### 3.3 Extrapolation of Compute-Optimal Scaling Law

The scaling coefficients of Section 3.2 were fitted on ladders spanning $10^{18}$ to $10^{20}$ FLOPs. To test whether the exponent gap persists to larger scales through extrapolation, we train Untied-Grow at $8\times$ the largest fitted compute and ask whether the fitted law predicts the loss of that run. The law does: the run lands on the extrapolated curve, and the gain carries over to downstream performance.

We rerun the Untied-Grow and Vanilla ladders on FineWeb-Edu using the same recipe fitted on FineWeb. We find that the optimal recipes transfer from FineWeb to FineWeb-Edu (Appendix C). The gains in scaling exponents from Vanilla to Untied-Grow are similar across the two datasets, while switching from FineWeb to FineWeb-Edu yields a constant multiplicative improvement in downstream performance (Figure 18). For more results on different datasets, see Appendix C. Building on the downstream scaling law in Grattafiori et al. (2024), we develop a scaling law to predict downstream performance (Appendix C.2) and set our training target to match the downstream capability of GPT-3 13B as estimated by Karpathy (2026). Our scaling law predicts that Untied-Grow $d26$, a 7.4B model that starts at 5B parameters, can reach this target using the compute-optimal recipe.

Figure 3 (left) shows that the 7.4B run lands on, and in fact slightly below, the extrapolated curve, so the exponent improvement of Section 3.2 extends to larger scales. The same holds for the downstream scaling law: the run's answer NLL and CORE score land on the curves predicted from the small-scale fits (Figure 3, middle and right).

The 7.4B Untied-Grow model reaches 0.3865 CORE, on par with the GPT-3 13B reference of 0.3852 (Karpathy, 2026), at $1.23 \times 10^{21}$ FLOPs against $2.31 \times 10^{22}$ for GPT-3 13B (Brown et al., 2020). The two models were trained on different data, and the GPT-3 reference is an estimate from a separate evaluation pipeline, so the roughly $20\times$ gap is indicative rather than a controlled comparison.

The compute-efficiency advantage of Untied-Grow over Vanilla widens with scale, as an exponent improvement predicts. Using the fitted FineWeb-Edu laws, the multiplier is $1.6\times$ at $10^{20}$ FLOPs, the end of the measured ladders, $1.8\times$ at $1.23\times10^{21}$ FLOPs, the compute of the 7.4B run (Figure 3), and $2.7\times$ at $10^{25}$ FLOPs, the budget of a modern pretraining run. The compute-efficiency gains on downstream metrics are larger than those on validation loss. This is because, at the same validation loss, Untied-Grow has slightly better downstream performance (Figure 17b).

### 3.4 Computational Depth

We interpret the scaling gains through *computational depth*: the number of blocks that meaningfully influence the predictive distribution. Executed depth can be wasted when later blocks barely change the prediction, which is a known problem in pre-norm transformers (Sun et al., 2025; Liu et al., 2020). We measure usable depth with *KL effective depth* (nostalgebraist, 2020; Csordás et al., 2026): we find the first block after the KL peak whose prediction is within 2 nats of the final output distribution. This is only a proxy: later blocks may still matter, and unused intermediate blocks go undetected.

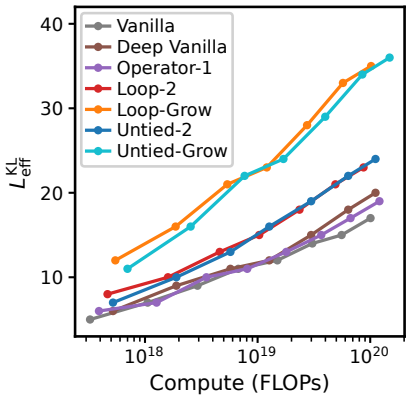



Figure 4: **KL effective depth over compute.**

Figure 4 shows three clusters of ladders. Increasing the number of core passes and growing the model improve the optimal KL effective depth. Notably, boundary operators improve the KL effective depth. Specifically, Untied-2 has a larger KL effective depth than Deep Vanilla, while the latter has only marginally greater KL effective depth than Vanilla. However, KL effective depth does not fully explain the efficiency gains we observe. For instance, untying weights has a small effect on KL effective depth, but a large one on loss. We emphasize that KL effective depth is an imperfect proxy for computational depths.

We have another two evidences showing that boundary operators improve the depth utilization. First, if we scale only the width, Untied-2 only has a constant improvement over Deep Vanilla, rather than an exponent improvement (Figure 15e). Second, if we choose width-to-depth ratio to be 64, depths becoming a more abundant resources, adding boundary operator doesn't bring a clear exponent improvement anymore (Figure 15c). These suggest that boundary operator improves exponents because there exists a deficient utilization of depths.

## 4 Data-Constrained Scaling

Section 3 showed that, on fresh data, neither weight sharing nor increasing the loop count across scales is compute-optimal. As compute grows faster than the supply of high-quality text (Villalobos et al., 2024), however, pretraining will increasingly reuse data over multiple epochs (Muennighoff et al., 2023; Kim et al., 2026; Lovelace et al., 2026; Vegesna et al., 2026). Repeated data lead to overfitting, which is typically mitigated by retuning weight decay at each model size (Kim et al., 2026). Recursion offers a different mitigation: adding depth without adding parameters to overfit. In this repeated-data regime, we find that the optimal loop count grows with compute, and tied weights become the better way to add capacity.

**Experimental setup.** To compare model-size scaling with loop-count scaling, we fix a pool of 100M unique FineWeb tokens and train every model for ten epochs, a budget of 1B tokens. We

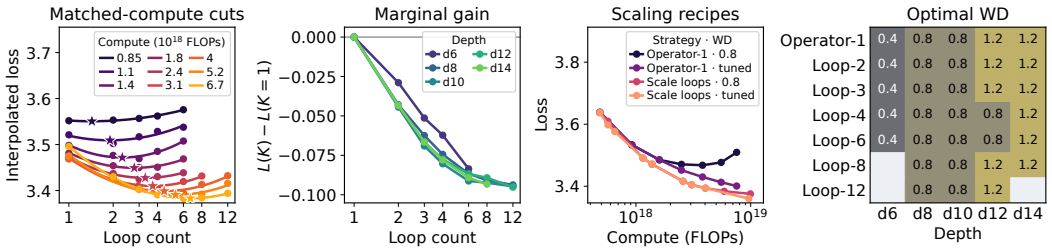



Figure 5: **In multi-epoch training, the optimal loop count grows with compute, scaling the loop count beats scaling model size even against tuned weight decay, and looping leaves the optimal weight decay nearly unchanged.** Models are trained on 100M unique FineWeb tokens for 10 epochs, and all losses shown are validation losses on FineWeb. **First: matched-compute cuts.** The loss of each loop count is interpolated in compute between neighbouring depths, curves are quadratics in log loss against log loop count, and stars mark their minima. The optimal loop count rises from 1.4 to 6.7 across budgets, and the gap to Operator-1 grows from 0.00 to 0.11. **Second: marginal gain from looping.** For each depth, we plot the loss at loop count $K$ minus the Operator-1 loss at the same depth. Every curve decreases monotonically, and from $d8$ upward the curves lie within 0.006 of one another at every loop count, so the gain from looping appears nearly independent of depth. **Third: weight decay when scaling loops versus model size.** Scaling model size at fixed weight decay overfits, and retuning weight decay at every model size only reaches 3.40 loss at $7.5 \times 10^{18}$ FLOPs. Scaling the loop count at fixed model size and fixed weight decay matches that loss with $2.2\times$ less compute, and tuning weight decay on top gains at most 0.02, so loop-count scaling is more compute-efficient and nearly free of tuning. **Fourth: optimal weight decay by depth and loop count.** The optimum rises with depth but is nearly flat in loop count, so overfitting tracks stored parameters rather than executed depth.

use the Operator-1 family from Section 3, with all hyperparameters other than weight decay fixed at the tuned values. The former adds parameters at a fixed loop count; the latter adds loops at a fixed parameter count. We train a grid over both, Operator-1 through Loop-12 at 120M to 1.4B parameters, and sweep weight decay over the same grid.

**Scaling the loop count.** We first examine how the preferred loop count changes with the compute budget. The optimal loop count increases with compute (Figure 5, first panel). We observe that every fixed-loop-count curve eventually overfits and turns upward, but the turn comes later and the minimum is lower for higher loop counts (Figure 1, upper right). At matched compute, the optimal loop count therefore rises from about one at the smallest budget to about seven at the largest, and the loss gap to Operator-1 grows from 0.00 to 0.11. Prior work shows that the optimal weight decay increases with parameter count on repeated data, because larger models overfit more and need more regularization (Kim et al., 2026). Looping appears to act as a similar regularizer: it delays overfitting without adding parameters, and its optimal strength similarly grows with compute.

**Controlling for weight decay.** We next test whether this advantage persists when model-size scaling receives its own regularization tuning. Scaling the loop count remains more compute-efficient than scaling model size, even when weight decay is tuned for model-size scaling (Figure 5, third panel). Here, scaling model size increases width and depth together at our fixed width–depth ratio. Retuning weight decay at every model size mitigates overfitting, but scaling the loop count at fixed model size and fixed weight decay matches the best tuned model-size-scaling loss with $2.2\times$ less compute. For loop-count scaling, however, additional weight-decay tuning reduces loss by at most 0.02, because the optimal weight decay of the Operator-1 model changes. This effect is visible in Figure 5 (fourth panel): the optimal weight decay rises with depth but is nearly flat in loop count, so the value tuned at one loop transfers to larger loop counts.

**Isolating weight sharing.** To separate the benefit of weight sharing from that of added depth, we compare against untied looping. It adds the same depth with more parameters and does not beat Operator-1 (Figure 23), so weight sharing is an effective regularization technique in multi-epoch training.

Together, these findings motivate the hypothesis that a model overfits with the parameters it stores, not with the blocks it executes. At small budgets the model is compute-limited, so a new parameter is preferred to a reused one, as on fresh data. Once the stored parameters begin to overfit the corpus, loops become the better way to add capacity, so the optimal loop count grows with compute. The second and fourth panels of Figure 5 support this interpretation: the marginal gains of looping are independent of the stored parameters and the optimal weight decay tracks stored parameters, not executed depth.

## 5 Related Work

**Scaling laws.** Neural scaling laws have been studied for some time (Kaplan et al., 2020; Hoffmann et al., 2022). A challenge remains: improvements at a fixed scale need not persist as compute grows (Rae et al., 2021). Maximal Update Parametrization (Yang et al., 2022) highlighted the importance of hyperparameter scaling in addressing training instability. Recent work has begun to show that interventions such as new optimizers (Liu et al., 2025; Qiu et al., 2026) and residual-stream designs (Chen et al., 2026) improve scaling constants. While some studies show that compute-efficiency gains increase at larger scales (Krajewski et al., 2024; Gloeckle et al., 2024), a lack of quantitative analysis of exponent improvements and explicit compute-optimal recipes makes direct comparison and interpretation difficult.

**Looped transformers and model growth.** Rather than compute efficiency, looped transformers have long been valued for their parameter efficiency (Prairie et al., 2026; Yang et al., 2024) and support for inference-time scaling (Geiping et al., 2025). Our results are consistent with these findings: we find that weight sharing is not a useful inductive bias in our natural-language training setting, but weight tying is more appealing in regimes where regularization is a concern, as seen in reasoning (Saunshi et al., 2025) and learning numerical algorithms (Yang et al., 2024). Schwethelm et al. (2026) show that looped transformers (with boundary operators) are not compute-efficient compared to standard transformer (without boundary operators). We suspect this is because they fix the block size of prelude and coda (Figure 14a and Figure 6).

Model growth reuses trained weights to initialize larger models (Chen et al., 2015; Du et al., 2024). Here, we study growth within a training run and examine its scaling frontier. McLeish et al. (2025) study the combination of growth and looped transformers in a different setting, converting pretrained models into looped ones. The most related concurrent work is SMELT (Wang et al., 2026c), which studies the compute-optimal scaling of sparse looped Mixture-of-Experts models. Our controlled comparisons isolate the effects of weight sharing, boundary operators, and growth.

**Computational depth scaling.** Evidence on the benefits of depth scaling is mixed (Kaplan et al., 2020; Tay et al., 2021; Liu et al., 2024; Wang et al., 2026b). HRM-Text (Wang et al., 2026a) demonstrates superior performance over transformers and attributes its success to overcoming the curse of depth (Csordás et al., 2026). We observe that boundary operators also improve KL effective depth. While the mechanisms—normalization and injection of earlier representations—are not entirely new (Sun et al., 2025; Loshchilov et al., 2025; Karpathy, 2025), to the best of our knowledge, we are the first to show their effects on scaling-law exponents. The role of computational depth in scaling laws remains an open question.

## 6 Conclusion

We have shown that model growth and boundary operators provide compute-efficiency gains that *increase* with scale. Furthermore, in the data-constrained setting, where we train for multiple epochs, increasing the loop count with scale outperforms standard model size scaling and weight decay tuning on the compute frontier. There are many fascinating directions for future work. In particular, architectural innovations that operationalize the notion of computational depth could lead to a new generation of foundation models with improved exponents. Within this effort, many standard design decisions can be reconsidered, spanning context length, the number of experts, width growth, and beyond. Moreover, carefully measuring the separate effects on scaling constants and scaling exponents deserves to become a key component of how architectural interventions are evaluated.

## AI USE STATEMENT

AI tools were used to assist with coding and provide writing feedback.

## REFERENCES

Yamini Bansal, Behrooz Ghorbani, Ankush Garg, Biao Zhang, Colin Cherry, Behnam Neyshabur, and Orhan Firat. Data scaling laws in nmt: The effect of noise and architecture. In *International Conference on Machine Learning*, pp. 1466–1482. PMLR, 2022.

Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. Language models are few-shot learners. *Advances in neural information processing systems*, 33:1877–1901, 2020.

Guangyu Chen, Yu Zhang, Jianlin Su, Weixin Xu, Siyuan Pan, Yaoyu Wang, Yucheng Wang, Guanduo Chen, Bohong Yin, et al. Attention residuals. *arXiv preprint arXiv:2603.15031*, 2026.

Tianqi Chen, Ian Goodfellow, and Jonathon Shlens. Net2net: Accelerating learning via knowledge transfer. *arXiv preprint arXiv:1511.05641*, 2015.

Róbert Csordás, Christopher D Manning, and Chris Potts. Do language models use their depth efficiently? *Advances in Neural Information Processing Systems*, 38:160313–160362, 2026.

Wenyu Du, Tongxu Luo, Zihan Qiu, Zeyu Huang, Yikang Shen, Reynold Cheng, Yike Guo, and Jie Fu. Stacking your transformers: A closer look at model growth for efficient llm pre-training. *Advances in Neural Information Processing Systems*, 37:10491–10540, 2024.

Leo Gao, Stella Biderman, Sid Black, Laurence Golding, Travis Hoppe, Charles Foster, Jason Phang, Horace He, Anish Thite, Noa Nabeshima, et al. The pile: An 800gb dataset of diverse text for language modeling. *arXiv preprint arXiv:2101.00027*, 2020.

Jonas Geiping, Sean McLeish, Neel Jain, John Kirchenbauer, Siddharth Singh, Brian Bartoldson, Bhavya Kailkhura, Abhinav Bhatele, and Tom Goldstein. Scaling up test-time compute with latent reasoning: A recurrent depth approach. *Advances in Neural Information Processing Systems*, 38: 41340–41391, 2025.

Fabian Gloeckle, Badr Youbi Idrissi, Baptiste Rozière, David Lopez-Paz, and Gabriel Synnaeve. Better & faster large language models via multi-token prediction. *arXiv preprint arXiv:2404.19737*, 2024.

Aaron Gokaslan and Vanya Cohen. Openwebtext corpus. `http://Skylion007.github.io/OpenWebTextCorpus`, 2019.

Aaron Grattafiori, Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Alex Vaughan, et al. The llama 3 herd of models. *arXiv preprint arXiv:2407.21783*, 2024.

Joel Hestness, Sharan Narang, Newsha Ardalani, Gregory Diamos, Heewoo Jun, Hassan Kianinejad, Md Mostofa Ali Patwary, Yang Yang, and Yanqi Zhou. Deep learning scaling is predictable, empirically. *arXiv preprint arXiv:1712.00409*, 2017.

Jordan Hoffmann, Sebastian Borgeaud, Arthur Mensch, Elena Buchatskaya, Trevor Cai, Eliza Rutherford, Diego de Las Casas, Lisa Anne Hendricks, Johannes Welbl, Aidan Clark, Tom Hennigan, Eric Noland, Katie Millican, George van den Driessche, Bogdan Damoc, Aurelia Guy, Simon Osindero, Karen Simonyan, Erich Elsen, Jack W. Rae, Oriol Vinyals, and Laurent Sifre. Training compute-optimal large language models. In *Advances in Neural Information Processing Systems*, 2022.

Jared Kaplan, Sam McCandlish, Tom Henighan, Tom B. Brown, Benjamin Chess, Rewon Child, Scott Gray, Alec Radford, Jeffrey Wu, and Dario Amodei. Scaling laws for neural language models. *arXiv preprint arXiv:2001.08361*, 2020.

Andrej Karpathy. Karpathy/nanochat: The best ChatGPT that $100 can buy., 2025. URL https://github.com/karpathy/nanochat.

Andrej Karpathy. [jan 7 2026] nanochat miniseries v1 · karpathy nanochat, 2026. URL https://github.com/karpathy/nanochat/discussions/420.

Konwoo Kim, Suhas Kotha, Percy Liang, and Tatsunori Hashimoto. Pre-training under infinite compute. In *International Conference on Learning Representations*, volume 2026, pp. 74596–74636, 2026.

Jakub Krajewski, Jan Ludziejewski, Kamil Adamczewski, Maciej Pióro, Michał Krutul, Szymon Antoniak, Kamil Ciebiera, Krystian Król, Tomasz Odrzygóźdź, Piotr Sankowski, et al. Scaling laws for fine-grained mixture of experts. *arXiv preprint arXiv:2402.07871*, 2024.

Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. Deep learning. *nature*, 521(7553):436–444, 2015.

Jeffrey Li, Alex Fang, Georgios Smyrnis, Maor Ivgi, Matt Jordan, Samir Gadre, Hritik Bansal, Etash Guha, Sedrick Keh, Kushal Arora, et al. Datacomp-lm: In search of the next generation of training sets for language models. *Advances in Neural Information Processing Systems*, 37:14200–14282, 2024.

Jingyuan Liu, Jianlin Su, Xingcheng Yao, Zhejun Jiang, Guokun Lai, Yulun Du, Yidao Qin, Weixin Xu, Enzhe Lu, Junjie Yan, et al. Muon is scalable for llm training. *arXiv preprint arXiv:2502.16982*, 2025.

Liyuan Liu, Xiaodong Liu, Jianfeng Gao, Weizhu Chen, and Jiawei Han. Understanding the difficulty of training transformers. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pp. 5747–5763, 2020.

Zechun Liu, Changsheng Zhao, Forrest Iandola, Chen Lai, Yuandong Tian, Igor Fedorov, Yunyang Xiong, Ernie Chang, Yangyang Shi, Raghuraman Krishnamoorthi, et al. Mobilellm: Optimizing sub-billion parameter language models for on-device use cases. *arXiv preprint arXiv:2402.14905*, 2024.

Ilya Loshchilov, Cheng-Ping Hsieh, Simeng Sun, and Boris Ginsburg. ngpt: Normalized transformer with representation learning on the hypersphere. In *International Conference on Learning Representations*, volume 2025, pp. 74014–74038, 2025.

Justin Lovelace, Christian Belardi, Srivatsa Kundurthy, Shriya Sudhakar, and Kilian Q Weinberger. Prescriptive scaling laws for data constrained training. *arXiv preprint arXiv:2605.01640*, 2026.

Sean McLeish, Ang Li, John Kirchenbauer, Dayal Singh Kalra, Brian R Bartoldson, Bhavya Kailkhura, Avi Schwarzschild, Jonas Geiping, Tom Goldstein, and Micah Goldblum. Teaching pretrained language models to think deeper with retrofitted recurrence. *arXiv preprint arXiv:2511.07384*, 2025.

Will Merrill and Ashish Sabharwal. A little depth goes a long way: The expressive power of log-depth transformers. *Advances in Neural Information Processing Systems*, 38:95315–95339, 2026.

Bruno Mlodozeniec, Pierre Ablin, Louis Béthune, Dan Busbridge, Michal Klein, Jason Ramapuram, et al. Completed hyperparameter transfer across modules, width, depth, batch and duration. In *International Conference on Learning Representations*, volume 2026, pp. 45300–45325, 2026.

Niklas Muennighoff, Alexander M Rush, Boaz Barak, Teven Le Scao, Aleksandra Piktus, Nouamane Tazi, Sampo Pyysalo, Thomas Wolf, and Colin Raffel. Scaling data-constrained language models. In *Advances in Neural Information Processing Systems*, 2023.

nostalgebraist. Interpreting gpt: The logit lens, 2020. URL https://www.lesswrong.com/posts/AcKRB8wDpdaN6v6ru/interpreting-gpt-the-logit-lens.

Dwarkesh Patel and Jerry Han, Sep 2026. URL https://www.dwarkesh.com/p/pretraining-progress-is-mostly-data.

Guilherme Penedo, Hynek Kydlíček, Anton Lozhkov, Margaret Mitchell, Colin Raffel, Leandro Von Werra, Thomas Wolf, et al. The fineweb datasets: Decanting the web for the finest text data at scale. *Advances in Neural Information Processing Systems*, 37:30811–30849, 2024.

Jackson Petty, Sjoerd van Steenkiste, Ishita Dasgupta, Fei Sha, Dan Garrette, and Tal Linzen. The impact of depth on compositional generalization in transformer language models. In *Proceedings of the 2024 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies (Volume 1: Long Papers)*, pp. 7239–7252, 2024.

Hayden Prairie, Zachary Novack, Taylor Berg-Kirkpatrick, and Daniel Y Fu. Parcae: Scaling laws for stable looped language models. *arXiv preprint arXiv:2604.12946*, 2026.

Shikai Qiu, Zixi Chen, Hoang Phan, Qi Lei, and Andrew Wilson. Hyperparameter transfer enables consistent gains of matrix-preconditioned optimizers across scales. *Advances in Neural Information Processing Systems*, 38:130867–130911, 2026.

Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, Ilya Sutskever, et al. Language models are unsupervised multitask learners. *OpenAI blog*, 1(8):9, 2019.

Jack W Rae, Sebastian Borgeaud, Trevor Cai, Katie Millican, Jordan Hoffmann, Francis Song, John Aslanides, Sarah Henderson, Roman Ring, Susannah Young, et al. Scaling language models: Methods, analysis & insights from training gopher. *arXiv preprint arXiv:2112.11446*, 2021.

Nikunj Saunshi, Nishanth Dikkala, Zhiyuan Li, Sanjiv Kumar, and Sashank J Reddi. Reasoning with latent thoughts: On the power of looped transformers. In *International Conference on Learning Representations*, volume 2025, pp. 14855–14881, 2025.

Kristian Schwethelm, Daniel Rueckert, and Georgios Kaissis. How much is one recurrence worth? iso-depth scaling laws for looped language models. *arXiv preprint arXiv:2604.21106*, 2026.

Wenfang Sun, Xinyuan Song, Pengxiang Li, Lu Yin, Yefeng Zheng, and Shiwei Liu. The curse of depth in large language models. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2025.

Yi Tay, Mostafa Dehghani, Jinfeng Rao, William Fedus, Samira Abnar, Hyung Won Chung, Sharan Narang, Dani Yogatama, Ashish Vaswani, and Donald Metzler. Scale efficiently: Insights from pre-training and fine-tuning transformers. *arXiv preprint arXiv:2109.10686*, 2021.

Matus Telgarsky. Benefits of depth in neural networks. In *Conference on learning theory*, pp. 1517–1539. PMLR, 2016.

Akshay Vegesna, Samip Dahal, Chinmay Karkar, Bishwas Mandal, Shmuel Berman, and Zhiwei Xu. Slowrun: Language modeling with infinite compute, fixed data. `https://github.com/qlabs-eng/slowrun`, 2026.

Pablo Villalobos, Anson Ho, Jaime Sevilla, Tamay Besiroglu, Lennart Heim, and Marius Hobbhahn. Will we run out of data? limits of llm scaling based on human-generated data, 2024. URL `https://arxiv.org/abs/2211.04325`.

Guan Wang, Changling Liu, Chenyu Wang, Cai Zhou, Yuhao Sun, Yifei Wu, Shuai Zhen, Luca Scimeca, and Yasin Abbasi Yadkori. Hrm-text: Efficient pretraining beyond scaling. *arXiv preprint arXiv:2605.20613*, 2026a.

Kevin Wang, Ishaan Javali, Michał Bortkiewicz, Tomasz Trzcinski, and Benjamin Eysenbach. 1000 layer networks for self-supervised rl: Scaling depth can enable new goal-reaching capabilities. *Advances in Neural Information Processing Systems*, 38:157643–157670, 2026b.

Shaowen Wang, Ge Zhang, Kairong Luo, Yuhao Wu, Shaofan Liu, Jiaheng Liu, Wenhao Huang, Shen Yan, and Jian Li. Smelt: Scaling laws for compute-matched moe looped transformers. *arXiv preprint arXiv:2609.01343*, 2026c.

Kaiyue Wen, David Hall, Tengyu Ma, and Percy Liang. Fantastic pretraining optimizers and where to find them. In *International Conference on Learning Representations*, volume 2026, pp. 144731–144838, 2026.

Greg Yang, Edward J. Hu, Igor Babuschkin, Szymon Sidor, Xiaodong Liu, David Farhi, Nick Ryder, Jakub Pachocki, Weizhu Chen, and Jianfeng Gao. Tensor programs v: Tuning large neural networks via zero-shot hyperparameter transfer, 2022. URL https://arxiv.org/abs/2203.03466.

Liu Yang, Kangwook Lee, Robert Nowak, and Dimitris Papailiopoulos. Looped transformers are better at learning learning algorithms. In *International conference on learning representations*, volume 2024, pp. 42195–42214, 2024.

## APPENDIX OUTLINE

The appendix provides the experimental details and additional results supporting the main text.

**Appendix A** describes the common training setup and architecture definitions, including the training procedure for Untied-Grow. It presents the sweeps used to choose block allocation, core-pass count, growth target, and growth timing, together with tests of token allocation and recipe sensitivity. It then gives the full recipe-fitting procedure: tuning a base model, selecting the token allocation and growth fraction, fitting the learning-rate scaling rule, and training the scaling ladders. The settings used for each architecture are listed in Table 10.

**Appendix B** presents additional compute-optimal scaling results and ablations. These isolate the components of the boundary operator, test block allocation, and compare efficiency in FLOPs and training time. Further experiments examine architecture-specific hyperparameter tuning, model shape, width-only scaling, random recurrence, test-time core passes, and optimizer choice, showing how these decisions affect constant-factor and exponent improvements.

**Appendix C** covers downstream evaluation and additional datasets. It specifies the DCLM CORE accuracy and answer-NLL protocols, including the fixed task subset, and compares downstream scaling, performance at matched validation loss, and category-level accuracy. It then describes the taskwise scaling laws used to forecast CORE accuracy and plan the large-model extrapolation, followed by recipe transfer and scaling comparisons between FineWeb and FineWeb-Edu. Finally, it compares Vanilla and Untied-Grow on OpenWebText, deduplicated Pile, and FineWeb, examining how architecture gains and downstream scaling vary across corpora and between DCLM CORE and OLMES.

**Appendix D** extends the data-constrained experiments across data-repetition levels, weight decay, and architectural controls. It examines how repetition changes the preferred number of core passes, how weight decay and recurrence interact, and how hyperparameters transfer across recurrence counts. Comparisons of tied and untied models, with and without the boundary operator, distinguish the roles of weight sharing and additional depth.

**Limitations and Future Directions.** We hold batch size, vocabulary size, and context length fixed. Studying how architecture interacts with these hyperparameters is an interesting next step. In particular, scaling computational depth and chain-of-thought reasoning can be complementary. The optimal trade-off between depth and reasoning length is an interesting question for future work. Our growth schedule is simple by design. We expect more sophisticated schedules to enable even greater exponent improvements. Combining our techniques with mixture-of-experts models is also an interesting next step. Although we conduct extensive ablation experiments, we do not provide a rigorous justification for why and how scaling exponents improve.

**Hardware.** Most runs use one node with eight NVIDIA H100 80GB HBM3 GPUs. The largest $d26$ extrapolation run uses two such nodes.

# A Experiment Details for Compute-Optimal Scaling

We explain the training protocols we use for Sections 3 and 4. The common setup is in Table 2.

Table 2: Fixed architecture and training protocol. Architecture-specific hyperparameters and token budgets are reported in the following subsections.

| Setting | Protocol |
|---|---|
| Context length | 2048 tokens |
| Tokenizer | GPT-2 vocabulary of 50,257, padded to 50,304 |
| Global batch | 524,288 tokens |
| Shape rule | Model name $d\ell$ denotes nominal/reference depth $\ell$ and width $w = 128\ell$ |
| Optimizer | Muon for matrix parameters; AdamW for embeddings and the language-model head. |

## A.1 Architecture

All models use the pre-norm decoder-only transformer, with RoPE, SwiGLU, and QK normalization. We use no biases or learned normalization gains, so trainable matrices are all two-dimensional. Additional RMS normalizations follow the token embedding and precede `lm_head`. Attention and MLP output projections and `lm_head` are initialized to zero; the token embedding is initialized normally, while the attention Q/K/V and MLP input matrices use uniform initialization.

Following Karpathy (2025), $d\ell$ names a model by the stored depth $\ell$, with an exception for Untied families. For Untied families, stored depths are $\ell + C$, where $C$ is the number of core blocks. Parameters for different architectures and reference depths are shown in Table 3.

Algorithm 1 gives the training procedure for Untied-Grow, including the boundary operator and the copying of core weights at the growth step.

**Algorithm 1** Untied Grow: normalize and re-inject with untied core growth

**Require:** Training batches $\{(x_t, y_t)\}_{t=1}^{T}$; injection scale $\alpha$
**Require:** Growth step $g$; initial core count $K_0$; final core count $K_f = mK_0$, $m \geq 2$
Initialize prelude $\mathcal{P}$, coda $\mathcal{C}$, and independent cores $\{\mathcal{R}_{\theta_k}\}_{k=1}^{K_0}$
$K \leftarrow K_0$
**for** $t = 1, \ldots, T$ **do**
  **if** $t = g + 1$ **then** ▷ Grow after $g$ training steps
    **for** $r = 1, \ldots, m-1$ **do**
      **for** $j = 1, \ldots, K_0$ **do**
        $\theta_{rK_0+j} \leftarrow \mathrm{copy}(\theta_j)$ ▷ Stack a copy of the core
      **end for**
    **end for**
    $K \leftarrow K_f$ ▷ Activate new untied cores
  **end if**
  $e \leftarrow \mathcal{P}(x_t), \quad h \leftarrow e$
  **for** $k = 1, \ldots, K$ **do**
    $h \leftarrow \mathcal{R}_{\theta_k}(h)$ ▷ Distinct weights for every pass
    $h \leftarrow \mathrm{RMSNorm}(h) + \alpha e$ ▷ After each core, including the last
  **end for**
  $\hat{y}_t \leftarrow \mathcal{C}(h)$
  $\mathcal{L}_t \leftarrow \mathrm{CrossEntropy}(\hat{y}_t, y_t)$
  Update all active parameters using $\nabla \mathcal{L}_t$ ▷ Backpropagate through all $K$ passes
**end for**
**return** Trained model

**Compute budget estimation** Plotted compute comes from the model FLOP estimator, which includes matrix multiplications and attention computation; expressions of the form $6TN_{\mathrm{eff}}$ below are leading-order allocation identities.

Table 3: **Model depths, layer splits, executed depths, and stored parameters.** P/C/D gives the prelude/core/coda layer split. Executed depth shows the transition from $K = 2$ to $K = 4$; fixed Loop-2 and Untied-2 use the first value and Loop-Grow and Untied-Grow use the second. Vanilla has executed depths equal to physical depths. Operator-1 and Loop-2 have the same stored parameters as Vanilla, while Loop-2 has the same effective parameters as Untied-2. Loop-Grow shares the Vanilla stored parameters, whereas Untied-Grow allocates all four untied core copies from the start.

| Depths | P/C/D | Exec. Depth | Width | Vanilla | Loop-Grow | Untied-2 | Untied-Grow |
|---|---|---|---|---|---|---|---|
| $d6$ | 2/2/2 | $8 \to 12$ | 768 | 120 M | 120 M | 130 M | 160 M |
| $d8$ | 2/3/3 | $11 \to 17$ | 1,024 | 210 M | 210 M | 240 M | 320 M |
| $d10$ | 3/4/3 | $14 \to 22$ | 1,280 | 330 M | 330 M | 410 M | 580 M |
| $d12$ | 4/4/4 | $16 \to 24$ | 1,536 | 490 M | 490 M | 610 M | 830 M |
| $d14$ | 4/5/5 | $19 \to 29$ | 1,792 | 730 M | 730 M | 920 M | 1.3 B |
| $d16$ | 5/6/5 | $22 \to 34$ | 2,048 | 1.0 B | 1.0 B | 1.3 B | 2.0 B |
| $d18$ | 6/6/6 | $24 \to 36$ | 2,304 | 1.4 B | 1.4 B | 1.8 B | 2.5 B |
| $d20$ | 6/7/7 | $27 \to 41$ | 2,560 | 1.8 B | 1.8 B | 2.4 B | 3.5 B |
| $d22$ | 7/8/7 | $30 \to 46$ | 2,816 | 2.4 B | 2.4 B | 3.2 B | 4.7 B |
| $d24$ | 8/8/8 | $32 \to 48$ | 3,072 | 3.0 B | 3.0 B | 3.9 B | 5.7 B |
| $d26$ | 8/9/9 | $35 \to 53$ | 3,328 | 3.8 B | 3.8 B | 5.0 B | 7.4 B |

**Multipliers** Suppose $\Theta$ is a parameter and $F$ is a layer (e.g., an MLP or attention layer). A multiplier $\alpha$ for $\Theta$ is applied as $F(\alpha\Theta)$. The multiplier is not trainable. Multipliers have an equivalence relationship with learning rates and initialization scales, known as ABC-parameterization (Yang et al., 2022). In this paper, we define two multipliers of interest. The output multiplier (OM) is the multiplier for the unembedding or readout layer. The residual multiplier (RM) is the shared weight multiplier for the MLP down projection and attention output projection.

### A.2 Compute-Optimal Decisions for Looping

Looped models introduce four choices beyond the base Transformer recipe: how to allocate blocks across the prelude, core, and coda; how many times to apply the core; how to increase that count during training; and how to allocate tokens after growth. We organize the evidence around these decisions.

#### A.2.1 Allocating Blocks across the Prelude, Core, and Coda

To identify the optimal allocation of core blocks as depth increases, we sweep the core size of tied, two-pass models at several stored depths, training each model for 1B tokens. Because a larger core executes more blocks per token, we compare losses against a common compute frontier rather than only within a fixed stored depth (Figure 6, left). The preferred core *fraction* varies little with depth, supporting a family whose three regions grow in proportion. Hence, we use a simple shared allocation: divide the blocks as evenly as possible across the prelude, core, and coda, assigning remainders first to the core and then to the coda.

#### A.2.2 Choosing the Number of Core Passes

**Fixed-token ladders favor one to two passes.** For each core-pass count $K \in \{1, 2, 3, 4, 6\}$, we train a ladder of model sizes on 1B tokens. We interpolate each ladder's loss at common compute budgets and fit loss against $K$. For both tied and untied models, the fitted optimum stays between one and two passes across the measured budgets; $K = 3, 4, 6$ give higher loss (Figure 7). We use $K = 2$ for the fixed-recurrence variants on this basis.

**At fixed model size, more compute favors more passes.** We next hold the compute budget and model depth fixed, sweep the same $K$ values, and reduce training tokens as $K$ increases to match compute. Across anchors $d8$–$d12$, the fitted optimal count $K^\star$ increases with the budget (Figure 8, left), consistent with Prairie et al. (2026). Thus, at fixed model size, larger budgets favor allocating some compute to extra passes.

Figure 8 (right) summarizes the optima across anchors using baseline tokens per stored parameter, $\mathrm{TPP}_{K1}$: the tokens affordable at $K = 1$ under the same budget, divided by that model's stored pa-

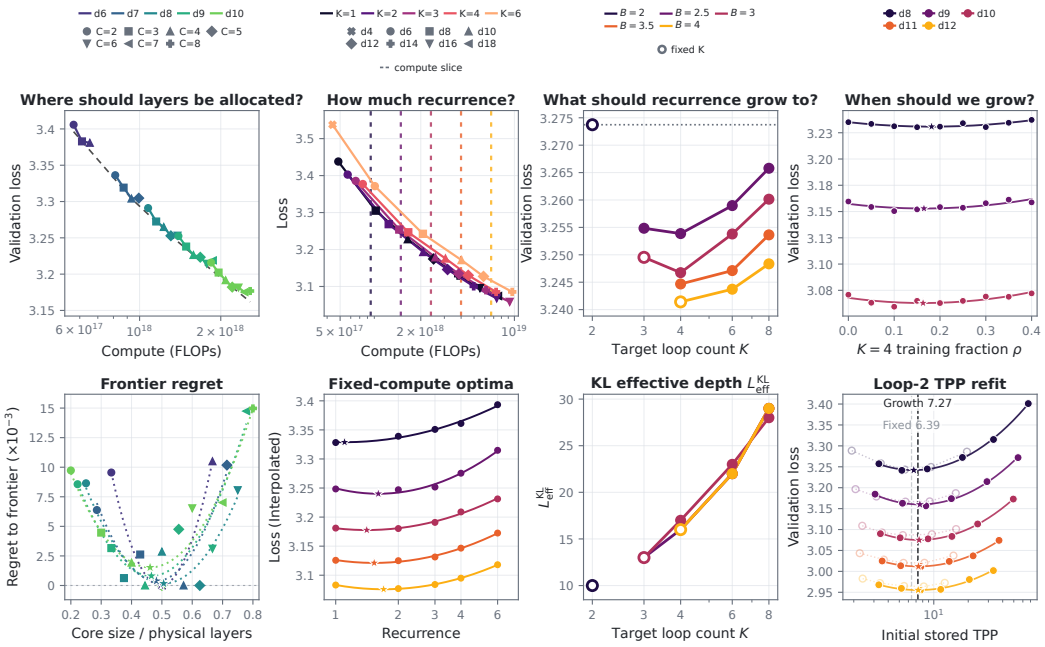



Figure 6: **Allocation, recurrence, and growth sweeps. Left:** loss versus compute for the 1B-token core-allocation sweep, followed by regret to the compute frontier versus core fraction; curves fit each depth's regret and stars mark their minima. **Middle left:** fixed-token recurrence ladders and fixed-compute slices interpolated in log loss versus log compute. **Middle right:** loss and logit-KL effective depth versus the growth target, with filled points for growth from $K = 2$ and hollow points for fixed-$K$ anchors. **Right:** Loop-2 growth-fraction sweeps at the depth-matched TPP-6 compute budgets, followed by the initial-stored-TPP refit; filled points use growth and hollow points are fixed controls. These panels summarize separate experiments with the controls described in the text.

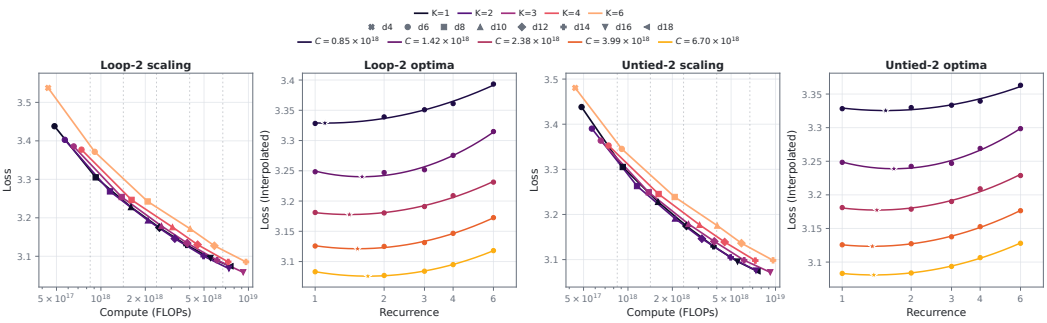



Figure 7: **Recurrence choice in fixed-token ladders.** Every model trains on 1B tokens with the Operator-1 base recipe. For each family, the loss–compute ladders are interpolated at common budgets, and quadratics in log recurrence and log loss locate the fitted optima (stars). The two families share the $K = 1$ ladder and are FLOPs-matched at each depth and recurrence.

rameter count. Define $N_{c,K}$ as the *compute-active parameter count*: the prelude, coda, and output-head matrix parameters counted once, plus the core matrix parameters counted $K$ times, even when shared. This count excludes the input embedding lookup; $N_{c,1}$ is the count for the same anchor with one core pass. The expansion is $E_K = N_{c,K}/N_{c,1}$. At $K^\star$, we fit

$$\begin{aligned} E_{K^\star} &= 0.82\,\mathrm{TPP}_{K1}^{0.15} \quad \text{(tied)}, \\ E_{K^\star} &= 0.83\,\mathrm{TPP}_{K1}^{0.15} \quad \text{(untied)}. \end{aligned} \tag{4}$$

The rightmost panels show that tokens per stored parameter at the optimum also increase with baseline TPP, but sublinearly: some of the additional budget goes to recurrence.

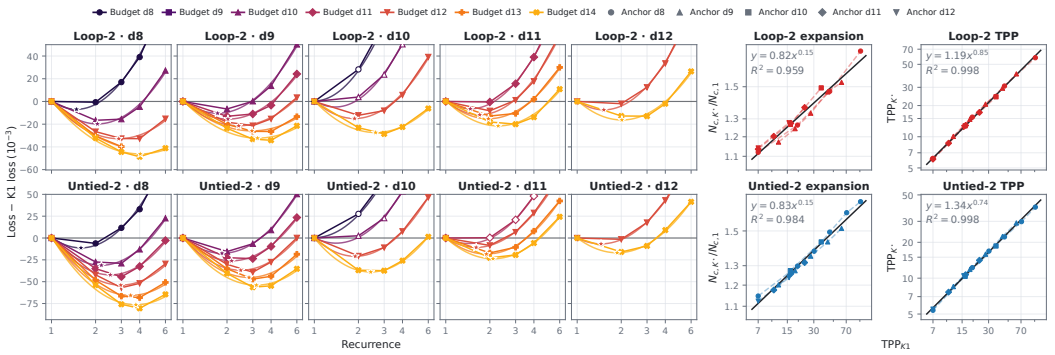



Figure 8: **Recurrence choice at fixed anchor size and compute.** The first row uses tied cores and the second uses untied cores. **Left:** loss relative to $K = 1$ versus recurrence, with anchor depth increasing across columns; each series holds compute fixed by adjusting tokens. Quadratic fits locate $K^\star$. **Right:** compute-active expansion and tokens per stored parameter at the fitted optimum versus the $K = 1$ baseline TPP. Hollow points mark optima outside the measured recurrence grid and are excluded from the fitted laws.

### A.2.3 Choosing the Growth Target and Transition

**Grow from two to four passes.** Starting from the tied $d8$ model with a $2/3/3$ block split, we sweep growth targets $K \in \{3, 4, 6, 8\}$ over 1B training tokens. At each compute budget, the transition is chosen to match the budget; thus a larger target is used for a smaller fraction of the run. Growing to four passes gives the lowest loss at every tested budget (Figure 6, middle right). Larger targets increase logit-KL effective depth without consistently improving loss. We adopt the $2 \to 4$ schedule for the growth variants: Loop-Grow reuses its tied core more times, while Untied-Grow duplicates the trained untied cores. Deep Vanilla Grow applies the same duplication schedule to plain Transformer blocks with the same prelude–core–coda allocation.

**Choose the transition at fixed depth and compute.** Let $\rho$ be the fraction of training tokens processed *after* growth, so a larger $\rho$ means an earlier transition. With the target fixed at four passes, we sweep $\rho$ at three anchor depths and six compute budgets for Loop-Grow, and at the same anchors with three budgets each for Untied-Grow. Tokens are adjusted to keep each depth–budget pair at fixed compute. Quadratic fits to validation loss locate $\rho^\star$. Larger budgets generally favor earlier growth, but the minima are broad (Figures 9a and 9b, top rows).

Across depths and budgets, the token allocation at the fitted optimum follows an affine relationship,

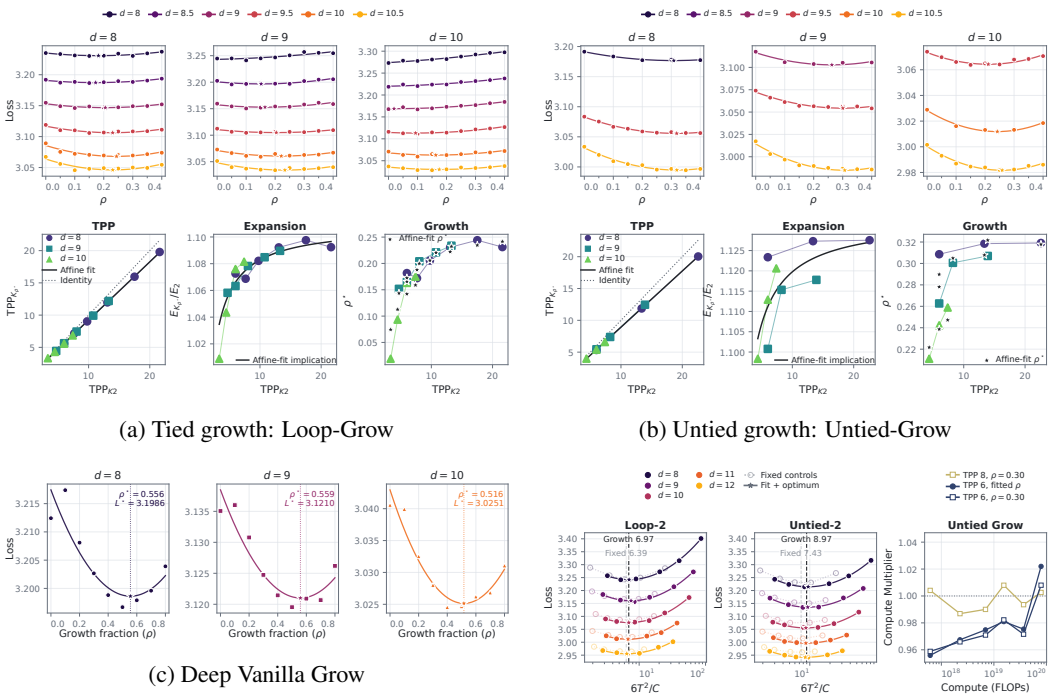
$$\mathrm{TPP}_{K_{\rho^\star}} = a\,\mathrm{TPP}_{K2} + b, \tag{5}$$

where both TPP quantities use the initial $K = 2$ stored parameter count: $\mathrm{TPP}_{K2}$ is the allocation without growth at the same compute, and $\mathrm{TPP}_{K_{\rho^\star}}$ is the allocation with the fitted transition. The coefficients are $(a, b) = (0.902, 0.216)$ for Loop-Grow and $(0.883, 0.103)$ for Untied-Grow, with $R^2 > 0.9999$ for both. This fit implies a compute expansion that rises and then plateaus with baseline TPP, motivating a nearly constant growth fraction at sufficiently high TPP. At the high-TPP end of the measured sweeps, the fitted fractions are approximately $0.23$–$0.24$ for Loop-Grow and $0.26$–$0.32$ for Untied-Grow. The exact conversion from expansion to $\rho$ depends slightly on the model's block allocation.

For Deep Vanilla Grow, we sweep one fixed-compute budget at each of $d8$, $d9$, and $d10$, using each anchor's fixed Deep Vanilla TPP-6 budget. The fitted fractions are $0.556$, $0.559$, and $0.516$, respectively, with similarly shallow minima near the midpoint of training (Figure 9c).

### A.2.4 Allocating Tokens and Testing Recipe Sensitivity

Growth increases average compute per token, so the optimal token allocation must be refitted. We repeat the iso-compute sweeps with growth in place (Figure 9d, left and middle). The figure uses $6T^2/C$, where $T$ is the number of training tokens and $C$ is training compute; under the leading-order relation $C = 6TN_{c,\mathrm{eff}}$, this coordinate equals tokens per training-averaged compute-active




(a) Tied growth: Loop-Grow

(b) Untied growth: Untied-Grow




(c) Deep Vanilla Grow




(d) Token allocation and growth prescription

Figure 9: **Growth timing, token allocation, and recipe sensitivity. (a,b)** Top rows show validation loss versus the post-transition token fraction $\rho$ at three model depths; colors identify fixed-compute budgets, curves are quadratic fits, and stars mark bracketed minima. Bottom rows show initial-stored-parameter TPP at the fitted transition, compute-active expansion, and optimal growth fraction versus fixed-$K = 2$ TPP. Colors and markers identify model depth; black curves and stars show the affine fit of Equation 5 and its implications. **(c)** Deep Vanilla Grow timing sweeps at each model's fixed Deep Vanilla TPP-6 budget, with fitted minima marked by stars. **(d)** Left and middle: loss versus $6T^2/C$ for tied and untied growth, where $T$ is the number of training tokens and $C$ is compute. Filled points use growth, hollow points are fixed controls, stars mark fitted minima, and dashed lines mark mean optima. Right: the Untied-Grow prescription ablation at even baseline depths from $d6$ to $d16$. The multiplier is the TPP-8, fitted-$\rho$ baseline's compute divided by each alternative's compute at matched loss, using the two nearest measured points in log-compute/log-loss space.

parameter. The mean optimum in this coordinate rises from 6.39 to 6.97 for tied growth and from 7.43 to 8.97 for untied growth.

We test the cost of simplifying both the token allocation and the transition for Untied-Grow and find that neither specification matters (Figure 9d, right). Thus, a simple recipe with a fixed growth fraction and TPP inherited from the non-growth version is near optimal.

### A.3 Compute-Optimal Recipe: Staged Sweeps and Fits

The recipe follows the four stages of Section 3.1: tune base hyperparameters at $d8$ (Stage 1), fit tokens per stored parameter (Stage 2), choose a fixed growth fraction $\rho$ for growth variants (Stage 3), and fit the learning-rate scaling rule (Stage 4). Fixed-recurrence models skip Stage 3. The block allocation, recurrence count, and growth target are chosen as described in Appendix A.2. For growth, we recommend retaining the Stage 2 TPP and calibrating $\rho$ once: the sensitivity tests in Appendix A.2.4 show little benefit from retuning TPP or refitting the transition across sizes.

### A.3.1 Stage 1: Tune a Base Model

We first tune every hyperparameter at the base size, $d8$, on a fixed budget of 1B tokens. We sweep one hyperparameter at a time and carry the local optimum into the next hyperparameter sweep, as in Wen et al. (2026). We show the hyperparameter grids in Table 4. Partial tuning trajectories are shown in Figure 10.

In addition to the architectures considered in Section 2.2, we consider a few ablation architectures. *"no-coda-inj"* replaces the boundary operator before the coda with normalization alone. *"+ norm"* and *"+ inject"* have only one component of the boundary operator. Table 5 lists the tuned losses of compute-matched arms and demonstrates the importance of both parts of the boundary operator. Also, the final boundary operator before the coda, which is missing in the looped transformer literature, improves the loss by a fair amount. In addition, we find that untying the weights has a benefit over tying them.

Tuning each architecture separately matters: transferring Vanilla's tuned recipe to the looped variants costs $(7.2\text{–}13.3) \times 10^{-3}$ in loss (Table 7), emphasizing the importance of separate tuning for each architecture.

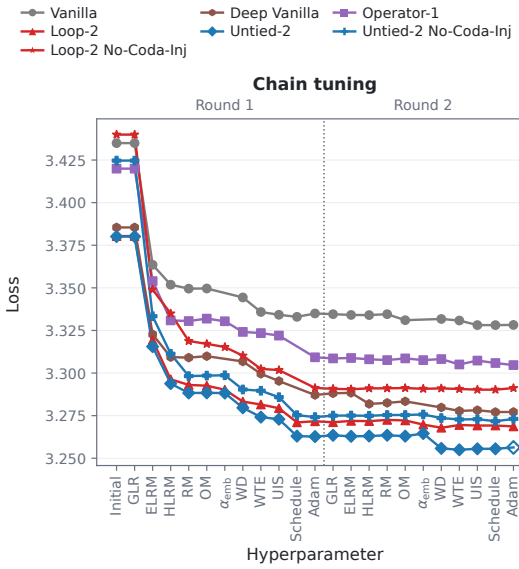



Figure 10: **Chain tuning at $d8$ on 1B tokens.** Each curve is one architecture's tuning chain. We adopt the new hyperparameters if they improve loss by $10^{-3}$ over the current best. Loss continuously drops with more tuning. ELRM, HLRM, WD, Schedule, WTE, and Adam bring most of the drops. The second round still improves loss for some HPs because the optimum changes as other HPs change. The loss plateaus near the end of the second round. The hollow marker denotes that more than one discrete HP improves performance and combining the changes can potentially improve performance further.

Table 4: **Tuning grid for chain tuning.** There are nine sweeps (eight for Vanilla) in each round. Each sweep has eight parallel runs. For numerical values, the grid is centered at the optimal values and uses factor-of-two spacing in the first round and factor-of-$\sqrt{2}$ spacing in the second round. For discrete values, there are two or three HPs. We change one HP in each run. If more than one HP change improves on the baseline, we apply the changes in the next sweep. Discrete sweeps use the same candidates in both rounds.

| HP | Definition | Initial HP |
|---|---|---|
| $\times\{1/16, 1/8, \ldots, 8\}$ | | |
| GLR | global learning rate | 0.04 |
| RM | residual multiplier | 0.5 |
| OM | output multiplier | 1 |
| $\alpha_{\text{emb}}$ | injection weight | 1 |
| $\times\{1/8, 1/4, \ldots, 16\}$ | | |
| WTE init. | embedding init. scale | 0.08 |
| UIS | input-matrix init. scale | 0.25 |
| $\times\{1, 2, \ldots, 128\}$ | | |
| ELRM | embedding LR multiplier | 0.02 |
| HLRM | head LR multiplier | 0.02 |
| $\{0\} \cup \times\{1/32, \ldots, 2\}$ | | |
| WD | weight decay | 0.1 |
| *Discrete* | | |
| Schedule | warmup $\{0, 5, 10, 20\}$; | 40; |
| | warmdown $\{.2, .6, .8, 1\}$ | .4 |
| Adam | $\beta_1\{.9, .95\}$; | $(.8, .95)$; |
| | $\beta_2\{.90, .98, .99\}$; | $10^{-10}$ |
| | $\epsilon\{10^{-8}, 10^{-6}\}$ | |

### A.3.2 Stage 2: Fit the Optimal Tokens per Parameter

With the base recipe frozen, we fit the split between model size and training tokens at fixed compute (Hoffmann et al., 2022). We express this split as tokens per stored parameter, $\text{TPP} = T/N$. Compute depends instead on the compute-active count $N_c$, which counts a shared core once per application (Appendix A.2.2). At the same anchor size and recurrence, Loop-2 and Untied-2 have the same compute per token, but Untied-2 stores more parameters; its stored-parameter TPP is therefore lower at matched compute. We use stored-parameter TPP throughout this stage.

Table 5: **Boundary-operator ablations at the base size.** Architectures and boundary maps follow Equation 2 and Equation 3. Each model is tuned independently and trained on 1B tokens at width 1024 and executed depth 11. Untied-2 achieves the lowest loss; removing either BO component or the coda injection increases loss. Runtime is the corresponding run's logged training time in minutes on eight H100 GPUs, excluding evaluation and initial compilation/warmup.

| Architecture | Tuned loss | Runtime (min) |
|---|---|---|
| Deep Vanilla | 3.2772 | 6.49 |
| Loop-2 | 3.2704 | 6.54 |
| Loop-2-no-coda-inj | 3.2912 | 6.45 |
| Untied-2 | **3.2563** | 6.54 |
| Untied-2-no-coda-inj | 3.2731 | 6.53 |
| Deep Vanilla + norm | 3.2781 | 6.48 |
| Deep Vanilla + injection | 3.2690 | 6.52 |

Table 6: **Base-tuned hyperparameters by architecture.** Recipes are selected by two rounds of chain tuning at the base size on 1B tokens. Dashes indicate inapplicable hyperparameters.

| Architecture | GLR | ELRM | HLRM | RM | OM | $\alpha_{\text{emb}}$ | WD | WTE | UIS | WU | WDR | $\beta_1$ | $\beta_2$ | $\epsilon$ |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Vanilla | 0.04 | 0.453 | 0.113 | 0.25 | 0.5 | — | 0.071 | 0.007 | 0.063 | 40 | 0.6 | 0.8 | 0.95 | $10^{-10}$ |
| Deep Vanilla | 0.04 | 0.16 | 0.057 | 0.5 | 1 | — | 0.1 | 0.005 | 0.5 | 5 | 0.8 | 0.8 | 0.99 | $10^{-8}$ |
| Operator-1 | 0.04 | 0.905 | 0.08 | 0.5 | 1 | 1 | 0.05 | 0.113 | 0.354 | 0 | 0.8 | 0.8 | 0.98 | $10^{-10}$ |
| Loop-2 | 0.04 | 0.32 | 0.113 | 0.25 | 1 | 0.707 | 0.05 | 0.02 | 0.044 | 40 | 1 | 0.8 | 0.95 | $10^{-10}$ |
| Untied-2 | 0.04 | 0.16 | 0.16 | 0.25 | 1 | 1 | 0.071 | 0.01 | 0.354 | 40 | 1 | 0.8 | 0.99 | $10^{-8}$ |

Table 7: **Transfer probe from the Vanilla recipe.** Each available transfer probe is trained once at $d8$ on 1B tokens with Vanilla's tuned recipe (injection variants keep the default $\alpha_{\text{emb}}$) and compared with its own tuned recipe. Transfer regret is the transferred-recipe loss minus the own-recipe loss, so positive values measure the benefit of variant-specific tuning; deltas and regrets use unrounded losses. Deep Vanilla's own-recipe loss is reported at matched executed depth 11 and width 1024. Vanilla's $-0.3 \times 10^{-3}$ is below the adoption threshold and reflects run-to-run noise.

| Architecture | Vanilla-recipe loss | $\Delta$ vs. Vanilla ($10^{-3}$) | Own-recipe loss | Transfer regret ($10^{-3}$) |
|---|---|---|---|---|
| Vanilla | 3.3275 | $+0.0$ | 3.3279 | $-0.3$ |
| Deep Vanilla | 3.2869 | $-40.6$ | 3.2772 | $+9.8$ |
| Operator-1 | 3.3135 | $-14.1$ | 3.3057 | $+7.8$ |
| Loop-2 | 3.2777 | $-49.9$ | 3.2704 | $+7.2$ |
| Untied-2 | 3.2697 | $-57.9$ | 3.2563 | $+13.3$ |

**Iso-compute sweeps.** We name each budget by the reference model size whose training at TPP 7 defines its compute cost. We use five budgets, $d8$ through $d12$. At each budget, we train five model sizes around the anchor and adjust the token count to match compute. We fit a quadratic in log loss against log TPP and take its vertex as the budget's optimal TPP (Figure 11). The vertex losses describe the fitted frontier under the base recipe. We defer scaling-law comparisons to the completed ladders, after fitting growth timing and learning-rate scaling.

**Is the optimal TPP stable across scale?** Hoffmann et al. (2022) found a roughly constant optimal TPP across compute, and we test the same property for each architecture. Table 8 reports the mean of the five vertices together with a scale-shift test. No variant rejects a scale-invariant TPP at $\alpha = 0.05$. We adopt the rounded mean in Table 8: 5 for Vanilla and 6 for Operator-1, Loop-2, and Untied-2. Every looped variant prefers a slightly higher TPP than Vanilla, indicating better parameter efficiency.

### A.3.3 Stage 3: Choose a Fixed Growth Fraction

For the $2 \to 4$ growth schedule, let $\rho$ denote the fraction of training tokens processed *after* growth; the transition therefore occurs after fraction $1 - \rho$. We recommend reusing the Stage 2 TPP to set the reference compute budget and choosing one fixed $\rho$ for the ladder. At each of three small-model anchors ($d8$–$d10$), sweep $\rho$ while adjusting tokens to preserve that anchor's compute budget, then fit

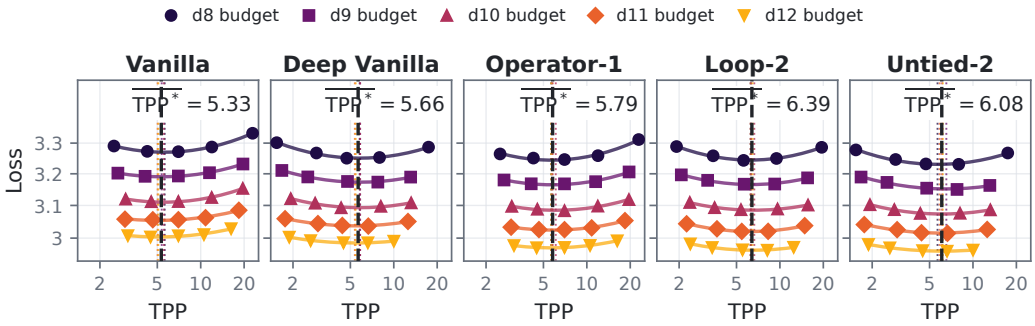



Figure 11: **Iso-compute TPP fits and fitted scaling laws.** Loss against tokens per parameter for Vanilla, Loop-2, and Untied-2 and a scaling law from fitted vertices. Each color is one fixed compute budget, the cost of the $d8$–$d12$ model at TPP 7; curves are quadratic fits of log loss against log TPP, dotted lines mark the per-budget vertices, and the dashed line is their mean. Per-budget vertices for the three architectures are plotted in the last subfigure as a scaling law.

Table 8: **Compute-optimal TPP by architecture.** Mean TPP$^\star$ averages the five iso-compute vertices; rounded TPP is the value used in the scaling ladders. The loss columns report the fitted vertex loss at budgets d8, d10, and d12; the best loss in each budget is bold. The scale-shift test fits $\log \mathrm{TPP}^\star = a + \beta \log C$ over all five budgets and reports the two-sided $p$-value of $H_0 : \beta = 0$. No variant rejects a scale-invariant TPP at $\alpha = 0.05$.

| Architecture | Mean TPP$^\star$ | Rounded | $p$ | $L^\star_{d8}$ | $L^\star_{d10}$ | $L^\star_{d12}$ |
|---|---|---|---|---|---|---|
| Vanilla | 5.33 | 5 | 0.087 | 3.270 | 3.110 | 3.003 |
| Deep Vanilla | 5.66 | 6 | 0.109 | 3.251 | 3.093 | 2.985 |
| Operator-1 | 5.79 | 6 | 0.923 | 3.244 | 3.085 | 2.971 |
| Loop-2 | 6.39 | 6 | 0.588 | 3.244 | 3.085 | 2.964 |
| Untied-2 | 6.08 | 6 | 0.760 | **3.230** | **3.073** | **2.961** |

a quadratic to validation loss versus $\rho$ (Figure 12). Average the three fitted optima and round to one decimal place. The broad minima support this simple choice without a growth-specific TPP refit or per-size transition rule.

We use this recipe for Deep Vanilla Grow: the mean optimum $0.544$ gives $\rho = 0.5$, with the Deep Vanilla TPP-6 reference budget. The reported Untied Grow and Loop Grow ladders instead use fitted per-size fractions from Appendix A.2.3 and refitted reference TPP values of $8$ and $7$, respectively (Table 10). We expect little benefit from these refinements (Appendix A.2.4).

### A.3.4 Stage 4: Fit the Learning-Rate Rule

With token allocation and growth timing fixed, we determine which base hyperparameters need to change with model size. We first screen Vanilla with one-dimensional sweeps over $d8$–$d12$, using the Stage 2 token allocation at each size (Figure 13a). All hyperparameters are swept at $d8$, $d10$, and $d12$; the intermediate sizes cover the global learning rate (GLR), head learning-rate ratio, output and residual multipliers, weight decay, and warmdown ratio. Scalar settings use $\{1/4, 1/2, 1, 2, 4\}$ times their base values, while schedule and Adam settings use discrete grids. The preferred GLR falls from $0.04$ at $d8$–$d10$ to $0.02$ at $d11$–$d12$, whereas several other settings have nearly flat loss curves.

We then sweep the global learning rate (GLR), output multiplier (OM), residual multiplier (RM), and weight decay (WD) at $d8$–$d10$ for the architecture families (Figure 13b). For each hyperparameter and size, we fit a quadratic to log validation loss against the log multiplier and take its minimum as the estimated optimum. Regressing the log optima against log stored parameter count gives the drift exponent $\beta$ in Table 9. The table also reports the loss penalty for a factor-of-two change and the full loss range of each sweep. Among these four hyperparameters, GLR has the largest sensitivity under

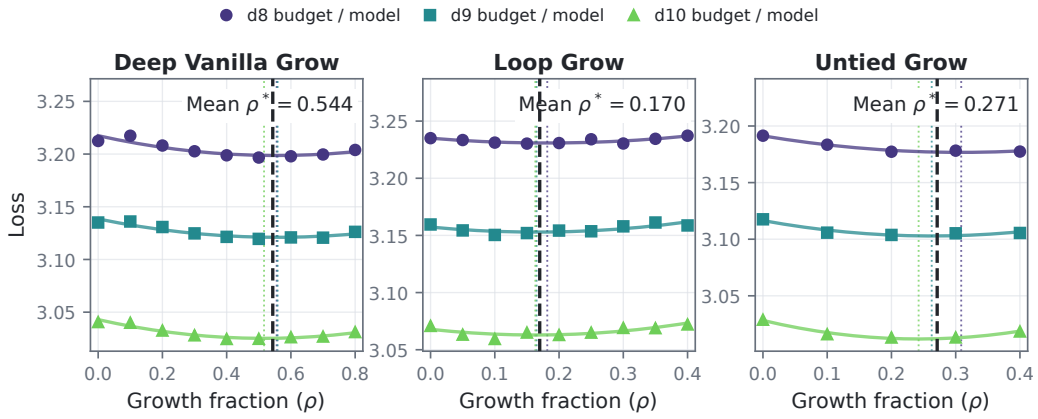



Figure 12: **Choosing a fixed growth fraction.** Left to right: Deep Vanilla Grow, Loop Grow, and Untied Grow. Each series shows validation loss versus the post-growth token fraction $\rho$ for a $d8$, $d9$, or $d10$ model at its corresponding fixed-compute budget; tokens are adjusted as $\rho$ varies. Points are measured runs and curves are quadratic fits. Colored dotted lines mark the fitted optima, and black dashed lines mark their arithmetic means: $0.544$, $0.170$, and $0.271$, respectively. The panels use different axis ranges.

the constant recipe. We therefore use the anchored learning-rate rule

$$\mathrm{GLR}(N) = \mathrm{GLR}_{d8} \left( \frac{N}{N_{d8}} \right)^{\beta}, \tag{6}$$

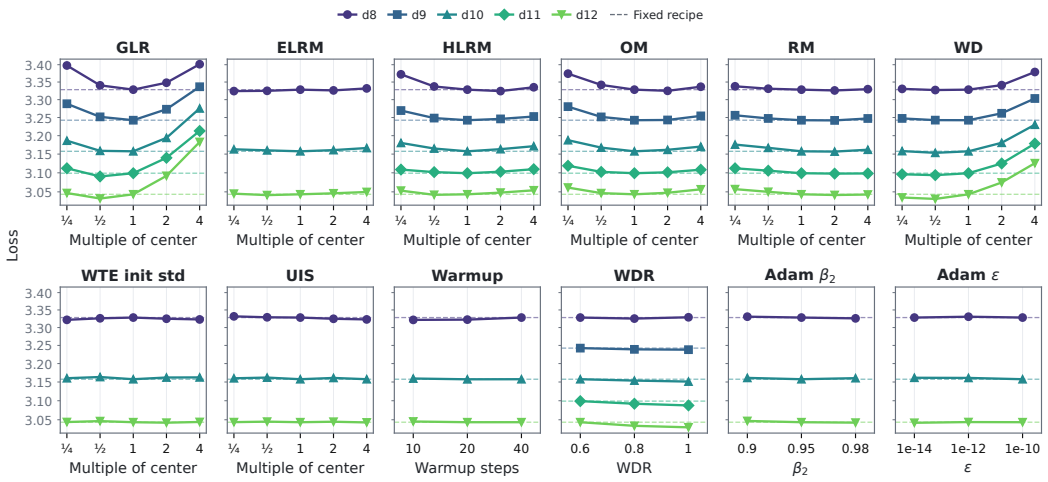
where $N$ is the initial stored parameter count and $N_{d8}$ is its value for that architecture's base model. The exponent is negative, so the learning rate decreases as models grow. For growth variants, we center the learning-rate sweeps on the corresponding fixed-recurrence rule and measure the remaining size dependence. The exponents and base learning rates used in the reported ladders are listed in Table 10; the exponents are specified to one decimal place.

We repeat the hyperparameter sweeps around the scaled recipe to check transfer across sizes (Figure 13b and Table 9). Scaling GLR also changes the preferred values of the other hyperparameters, so drift measured under a constant recipe need not imply that an additional scaling rule is useful. In the Vanilla ladder ablation, adding output-multiplier or weight-decay scaling changes the matched-loss compute multiplier by at most about $3\%$ relative to GLR scaling alone (Figure 15a). We therefore use the fitted GLR rule and keep the other hyperparameters at their base-tuned values. This gives a simple recipe for comparing architectures without carrying a separate scaling law for every hyperparameter.

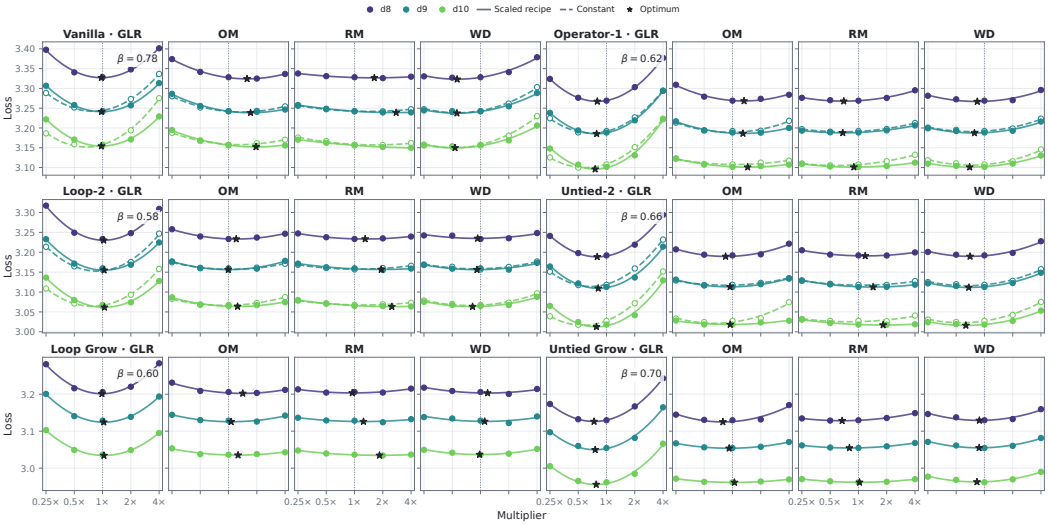
### A.3.5 Train the Ladders with the Fitted Recipe

Once the recipe is fixed, we train the scaling ladder. For a fixed-recurrence model, we count stored parameters $N$ and set the token budget to $T = \mathrm{TPP} \times N$. For a growth variant, the recommended recipe retains the Stage 2 allocation prescription and uses the fixed Stage 3 growth fraction, with tokens adjusted to preserve the prescribed compute budget across the two phases. We then apply the Stage 4 learning-rate rule and keep the other base hyperparameters unchanged. Table 10 lists the settings used for the reported ladders.

The ladders span $d6$–$d20$ for Vanilla (120M–1.8B stored parameters) and $d6$–$d18$ for the looped variants (120M–1.4B for K2 and 130M–1.8B for Dep; Table 3). Because the looped variants execute more blocks per token and train on more tokens per parameter, every ladder covers a similar range of compute, ending near $10^{20}$ FLOPs at its largest size despite the different stored-parameter counts. We compare at equal compute and fit Equation 1 to each ladder. First, we fit the irreducible loss $E$ from the Vanilla ladder with Huber loss minimization (Hoffmann et al., 2022). For the remaining ladders, we regress log reducible loss against log compute and report the exponent's regression standard error.




(a) Vanilla hyperparameter sensitivity across scale




(b) Architecture-specific sweeps under constant and scaled recipes

Figure 13: **Hyperparameter sensitivity and transfer across scale.** **(a)** One-dimensional Vanilla sweeps at $d8$–$d12$; dashed lines show the fixed-recipe loss. Scalar settings use $\{1/4, 1/2, 1, 2, 4\}$ times the base value; schedule and Adam settings use the displayed grids. ELRM and HLRM are embedding and head learning-rate ratios, WTE init std is the embedding initialization standard deviation, UIS is the uniform initialization scale, and WDR is the warmdown ratio. **(b)** Sweeps of global learning rate (GLR), output multiplier (OM), residual multiplier (RM), and weight decay (WD) at $d8$–$d10$. Rows pair Vanilla with Operator-1, Loop-2 with Untied-2, and Loop-Grow with Untied-Grow. Solid and dashed curves are quadratic fits for scaled and constant recipes; stars mark scaled-recipe optima. Positive $\beta$ annotations use GLR $\propto N^{-\beta}$, opposite to the signed convention in Equation 6. Fixed-recurrence panels report constant-recipe fits; growth panels show inherited rule magnitudes.

## B Additional Results for Compute-Optimal Scaling

Figures 14 and 15 test how the scaling results depend on architecture, training recipe, model shape, and optimizer. Figure 14 examines the boundary operator, runtime efficiency, and constant-recipe baselines; Figure 15 separates the effects of tuning, depth, and recurrence. We compare the compute needed to reach the same loss, using interpolation within the measured ladders. Each ablation's

Table 9: **Fitted hyperparameter drift and sensitivity.** GLR, OM, RM, and WD denote global learning rate, output multiplier, residual multiplier, and weight decay. Each $\beta$ is fitted from the optima of quadratic slices at $d8$–$d10$, with one regression standard error; bold entries have $|\beta| \geq 3$ standard errors. For constant recipes, the optimal value scales as $N^{\beta}$. For +GLR recipes, the GLR column measures residual drift relative to the applied exponent shown in the recipe column; the other columns still measure drift relative to constant base values. Regret is the mean loss of the $0.5\times$ and $2\times$ settings minus the center loss; range is the full loss span of the slice. Both are in units of $10^{-3}$ at the largest fully swept size. The $d8$ anchor is shared across constant and scaled recipes. Growth variants are swept around their fixed-recurrence GLR rule. These diagnostic fits are distinct from the deployed settings in Table 10.

| | | GLR | | OM | | RM | | WD | |
|---|---|---|---|---|---|---|---|---|---|
| Variant | Recipe | $\beta$ | regret / range | $\beta$ | regret / range | $\beta$ | regret / range | $\beta$ | regret / range |
| Vanilla | constant | $\mathbf{-0.78 \pm 0.06}$ | 18.7 / 150.0 | $\mathbf{-0.37 \pm 0.10}$ | 3.5 / 17.6 | $+0.06 \pm 0.37$ | 2.3 / 15.2 | $-0.17 \pm 0.11$ | 9.4 / 94.0 |
| | +GLR ($\beta$=−0.8) | $-0.00 \pm 0.01$ | 15.0 / 73.3 | $\mathbf{+0.46 \pm 0.05}$ | 5.9 / 39.8 | $\mathbf{+2.73 \pm 0.23}$ | 1.4 / 20.7 | $-0.11 \pm 0.05$ | 4.9 / 53.3 |
| Operator-1 | constant | $\mathbf{-0.62 \pm 0.02}$ | 17.9 / 122.4 | $-0.43 \pm 0.55$ | 3.0 / 14.4 | $-0.41 \pm 0.15$ | 3.0 / 26.9 | $-0.32 \pm 0.20$ | 5.6 / 37.8 |
| | +GLR ($\beta$=−0.6) | $\mathbf{-0.11 \pm 0.02}$ | 17.4 / 121.6 | $+0.19 \pm 0.17$ | 3.8 / 21.5 | $+0.53 \pm 0.35$ | 1.4 / 10.8 | $\mathbf{-0.39 \pm 0.05}$ | 5.4 / 28.6 |
| Loop-2 | constant | $\mathbf{-0.58 \pm 0.08}$ | 16.8 / 110.1 | $\mathbf{-0.60 \pm 0.04}$ | 2.3 / 14.9 | $-0.09 \pm 0.38$ | 2.1 / 11.6 | $\mathbf{-0.40 \pm 0.06}$ | 7.0 / 41.1 |
| | +GLR ($\beta$=−0.6) | $\mathbf{+0.03 \pm 0.00}$ | 12.2 / 71.5 | $+0.08 \pm 0.51$ | 3.0 / 21.9 | $\mathbf{+1.36 \pm 0.19}$ | 3.4 / 16.1 | $-0.26 \pm 0.10$ | 2.7 / 23.2 |
| Loop-Grow | +GLR ($\beta$=−0.6) | $+0.10 \pm 0.05$ | 12.2 / 66.3 | $-0.23 \pm 0.55$ | 1.4 / 16.9 | $\mathbf{+1.40 \pm 0.12}$ | 0.4 / 13.6 | $\mathbf{-0.41 \pm 0.06}$ | 3.9 / 15.5 |
| Untied-2 | constant | $\mathbf{-0.66 \pm 0.02}$ | 17.3 / 134.1 | $-0.53 \pm 0.26$ | 1.9 / 49.7 | $\mathbf{-0.73 \pm 0.11}$ | −0.4 / 14.2 | $\mathbf{-0.43 \pm 0.02}$ | 6.2 / 50.4 |
| | +GLR ($\beta$=−0.7) | $-0.04 \pm 0.07$ | 14.8 / 111.2 | $+0.20 \pm 0.13$ | 2.7 / 10.1 | $\mathbf{+0.84 \pm 0.01}$ | 1.7 / 12.8 | $\mathbf{-0.20 \pm 0.03}$ | 5.1 / 35.0 |
| Untied-Grow | +GLR ($\beta$=−0.7) | $\mathbf{+0.10 \pm 0.01}$ | 13.1 / 104.3 | $\mathbf{+0.53 \pm 0.06}$ | 1.6 / 9.8 | $\mathbf{+0.81 \pm 0.02}$ | 2.0 / 8.5 | $-0.12 \pm 0.04$ | 7.2 / 28.2 |

Table 10: **Recipes used for the reported ladders.** Exponents apply in Equation 6 with $\mathrm{GLR}_{d8} = 0.04$ for every family except Deep Vanilla Grow, whose fitted $d8$ anchor is $0.036$. TPP is $T/N$ over stored parameters; for the Grow variants, $t_{\mathrm{ref}} = T_0/N_{\mathrm{ref}}$ sets the pre-growth compute budget, with tokens recomputed after choosing the transition. Deep Vanilla Grow uses the Deep Vanilla TPP-6 reference budget and $\rho = 0.5$. The Loop Grow and Untied Grow experiments include refitted growth allocations; the sensitivity results motivate the simpler prescription of reusing the fixed-recurrence allocation and a constant $\rho$ (Appendix A.3.3). All other hyperparameters keep their base-tuned values.

| Architecture | GLR exponent $\beta$ | Token allocation | Measured models |
|---|---|---|---|
| Vanilla | $-0.8$ | TPP 5 | $d6$–$d20$, even |
| Deep Vanilla | $-0.7$ | TPP 6 | $d6$–$d18$, even |
| Operator-1 | $-0.6$ | TPP 6 | $d6$–$d20$, even |
| Loop-2 | $-0.6$ | TPP 6 | $d6$–$d18$, even |
| Untied-2 | $-0.6$ | TPP 6 | $d6$–$d18$, even |
| Deep Vanilla Grow | $-0.8$ | $t_{\mathrm{ref}} = 6$ | $d6$–$d18$, even |
| Untied Grow | $-0.6$ | $t_{\mathrm{ref}} = 8$ | $d6$–$d18$, even |
| Loop Grow | $-0.5$ | $t_{\mathrm{ref}} = 7$ | $d6$–$d18$, even |

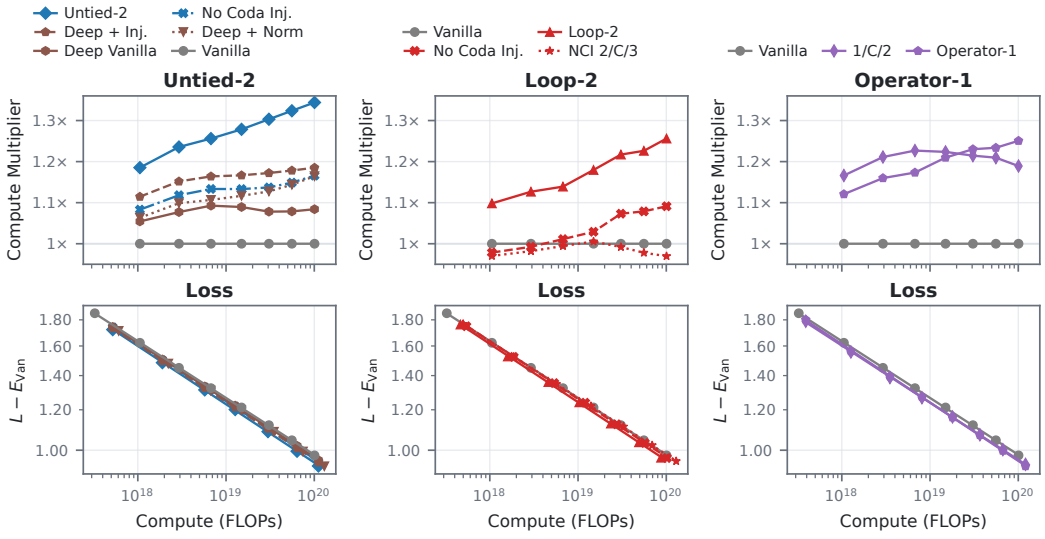
reference is specified in its caption, so multiplier magnitudes should not be compared directly across panels.

### B.1 Architecture Choices and Runtime Efficiency

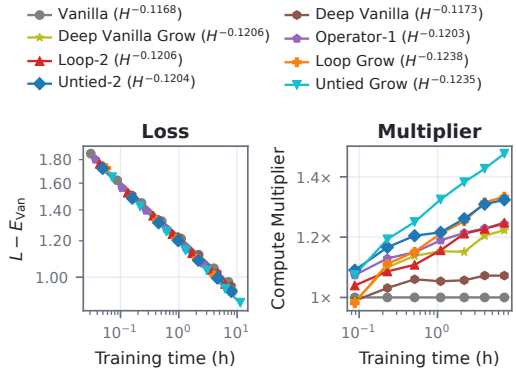
**The full boundary operator retains the largest gain.** Figure 14a (left) compares Untied-2 with plain Deep Vanilla and three operator ablations. Near $10^{20}$ FLOPs, Untied-2 reaches roughly $1.34\times$ the compute efficiency of Vanilla, while using only normalization or injection, or omitting coda injection, reduces the multiplier to about $1.17$–$1.19\times$. Deep Vanilla remains near $1.08\times$. Thus, extra executed depth alone does not recover the full gain, and each of these operator choices contributes over the measured range.

**Block allocation matters as models grow.** Removing coda injection also reduces Loop-2's gain (Figure 14a, middle). Fixing its prelude and coda at two and three blocks while scaling only the core is worse still: the multiplier falls below one at the largest budgets. Operator-1 shows a similar limitation with a fixed $1/C/2$ allocation: its early advantage peaks and then declines, while the proportional allocation continues improving (right). These comparisons support scaling the prelude, core, and coda together.

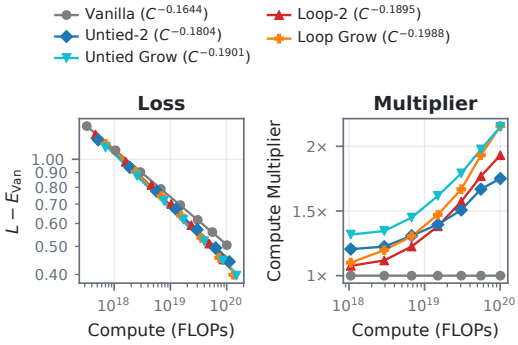
**The gains carry over to training time.** Figure 14b replaces FLOPs with recorded optimization time. The growth variants retain increasing time savings at matched loss, with Untied Grow giving the largest gain. The benefit therefore survives implementation costs on our hardware.




(a) Architecture ablations




(b) Major variants by runtime




(c) Constant recipes

Figure 14: **Architecture, runtime, and constant-recipe ladders.** Points are measurements; loss curves fit power laws above a shared Vanilla-fitted floor. **(a)** Columns compare Untied-2 boundary-operator ablations, Loop-2 coda and allocation ablations, and Operator-1 block allocations. Rows show compute multipliers and reducible loss. NCI denotes no coda injection; $2/C/3$ and $1/C/2$ fix the prelude and coda depths while scaling the core. **(b)** Major variants against recorded optimization time, excluding evaluation and checkpoint overhead; the multiplier is Vanilla time divided by each variant's time at matched loss. **(c)** Ladders with constant base recipes; the multiplier uses constant-recipe Vanilla as the reference. Multipliers interpolate between measurements in log compute or log time without extrapolation.

### B.2 Recipe Tuning Changes the Apparent Scaling Advantage

Constant base recipes favor the looped and grown models over Vanilla (Figure 14c). This is consistent with the ordering of $\beta$ magnitudes in Table 9. Vanilla has the most negative $\beta$, and Loop-2 has the least negative values of $\beta$. As a result, compute multipliers under constant recipes are better than those under scaled ones, and in the long run, Loop-Grow is the best constant recipe. This emphasizes the importance of hyperparameter optimality in scaling (Qiu et al., 2026; Mlodozeniec et al., 2026). Figure 15a reinforces this theme: scaling Vanilla's learning rate substantially improves its ladder. Adding output-multiplier or weight-decay scaling changes the multiplier by at most about 3% rela-

tive to GLR scaling alone, whereas the tested muP output-multiplier rule without GLR scaling loses efficiency with scale. This supports the GLR-only scaling prescription in Appendix A.3.4.

Tuning must also be architecture-specific. Simply ablating Operator-1's recipe with Vanilla's base-tuned hyperparameters causes the exponent improvements to become constant ones (Figure 15b). Transferring a baseline recipe can therefore hide an architecture's scaling improvement even when it remains better at individual budgets.

### B.3 Model Shape and the Direction of Scaling

Making a model deeper at a given width does not consistently improve its compute efficiency (Figure 15c). Deep Vanilla gives a modest gain over Vanilla, but the still-deeper $1{:}64$ depth-to-width ladders (Deeper series) do not consistently outperform their corresponding default shapes. In particular, Deeper Operator-1 loses the increasing advantage of Operator-1 over the measured range.

Holding executed depth fixed provides a more direct control (Figure 15e). At depth 11, width-only Untied-2 remains more efficient than Deep Vanilla, but Untied-2 now has a better constant, not exponent, than Deep Vanilla. This supports the idea that boundary operators become more useful as depth increases and the hypothesis that compute-optimal scaling exponents improve because of increased computational depth.

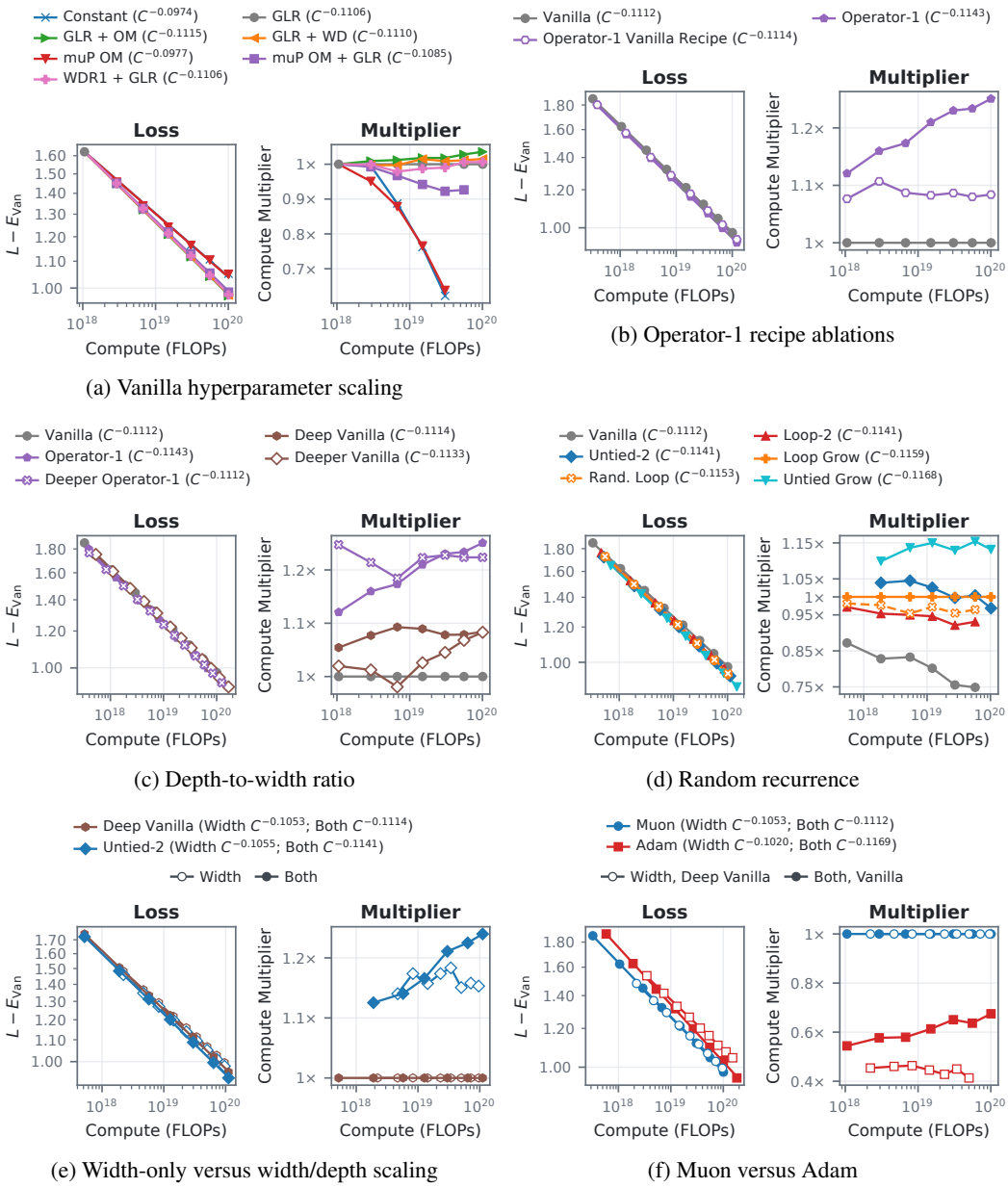
### B.4 Random Recurrence and Test-Time Passes

We match the Loop Grow recipe but sample $K$ uniformly from $\{2, 3, 4, 5, 6\}$ at each optimizer step after growth, preserving mean recurrence four. Compute accounting uses the realized recurrence counts, and evaluation uses $K = 4$. This random-recurrence ladder is slightly worse than fixed Loop Grow at matched compute (Figure 15d).

Random training does improve tolerance to extra test-time passes (Figure 16). At $k = 8$, loss relative to $k = 4$ changes by $-0.0003$ to $+0.0028$, compared with increases of $0.008$–$0.042$ for fixed Loop Grow. Five of seven random-recurrence checkpoints have a shallow minimum at $k = 5$, but the improvement is below $0.001$ loss and does not continue with further passes. In this setting, random recurrence chiefly reduces the penalty for extra passes rather than providing sustained test-time scaling.

### B.5 Optimizer Choice Affects Both the Constant and the Exponent

Muon has a constant improvement over Adam under width-only scaling, consistent with Qiu et al. (2026), but in coupled width/depth scaling, we find that compute efficiency gains *shrink* at larger scales (Figure 15f). This suggests that the Muon optimizer may not be as effective as Adam for deeper networks. This discrepancy between width-only and joint scaling motivates investigation into the interaction between architectures and optimizers.




(a) Vanilla hyperparameter scaling

(b) Operator-1 recipe ablations

(c) Depth-to-width ratio

(d) Random recurrence

(e) Width-only versus width/depth scaling

(f) Muon versus Adam

Figure 15: **Recipe, shape, recurrence, and optimizer ablations.** Each subfigure shows reducible loss and a loss-matched compute multiplier, interpolated without extrapolation. **(a)** Vanilla recipes share a fitted $d8$ anchor; multipliers are relative to constant + GLR. OM, WD, and WDR denote output multiplier, weight decay, and warmdown ratio. **(b)** Ablating Operator-1's recipe by using Vanilla's base-tuned hyperparameters. **(c)** Shape comparisons; Deeper Vanilla and Deeper Operator-1 use depth-to-width ratio 1:64. Panels (b,c) use Vanilla as the reference. **(d)** Random recurrence samples $K \in \{2, 3, 4, 5, 6\}$ after growth and evaluates at $K = 4$; compute uses the realized recurrence counts, with fixed Loop Grow as the reference. **(e,f)** Open markers indicate width-only scaling at executed depth 11; filled markers indicate coupled width/depth scaling. Each regime uses its own reference: Deep Vanilla with Muon in (e), and the corresponding Muon ladder in (f). Each optimizer uses its fitted token budget.

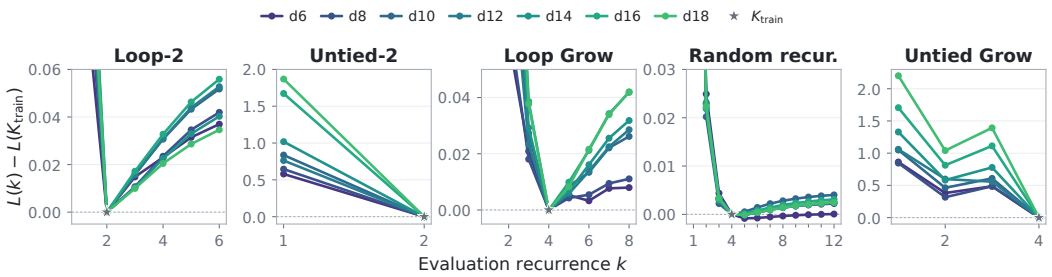



Figure 16: **Sensitivity to evaluation recurrence.** Loss differences $L(k) - L(K_{\text{train}})$ for the final-ladder checkpoints, with color indicating model size and stars marking the reference recurrence: $K_{\text{train}} = 2$ for Loop-2 and Untied-2, and 4 for the growth variants. Tied models reuse their core for additional passes; untied models can only run their allocated cores. Fixed Loop-2 and Loop Grow are best at their trained recurrence. Random-recurrence training produces much flatter curves, with small gains at $k = 5$ for five of seven checkpoints but no sustained improvement as more passes are added. Panels use different vertical scales.

# C Downstream Evaluation and Additional Datasets

We first discuss the downstream evaluation and scaling law from Section 3.3 and then present runs on additional datasets.

## C.1 Downstream Evaluations

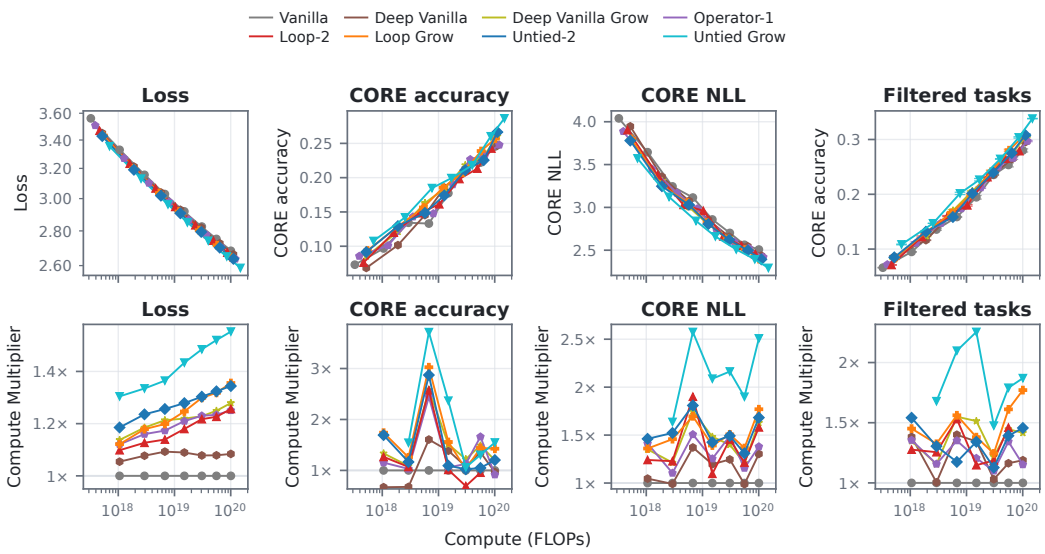
We evaluate all 91,037 examples in the 22-task DCLM CORE suite at each checkpoint. Our primary metrics are full CORE accuracy and answer negative log-likelihood (NLL); we also report accuracy on a fixed subset of 17 tasks.

**Full CORE accuracy.** CORE is the equally weighted mean of chance-normalized task accuracies (Li et al., 2024), with zero denoting chance and one denoting perfect accuracy. We follow the task definitions and scoring implementation in Karpathy (2025).

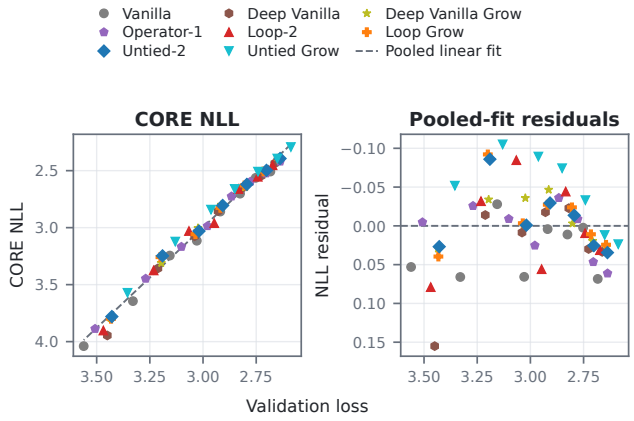
**Answer NLL.** Following Grattafiori et al. (2024), we also evaluate the mean token NLL of the reference answer, excluding prompt tokens. We average examples within each task and then average tasks equally. Lower values are better; the figures label this metric CORE NLL.

**Frozen Vanilla-only filter.** Some tasks provide little signal at the scales studied, so we additionally report mean centered accuracy on a subset selected using only the eight Vanilla checkpoints. Inspired by the evaluation-task selection criteria of Penedo et al. (2024), a task is retained if the largest-token checkpoint's raw accuracy is at least two percentage points above chance and Kendall's $\tau_b$ between training amount and accuracy is at least $0.50$. This excludes the five tasks in Table 11, leaving 17 tasks. The subset stays fixed across architectures, recipes, and corpora.

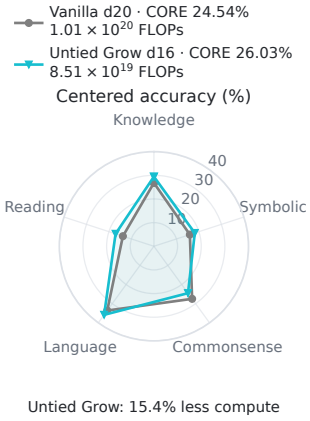
**Downstream Ladders.** We evaluate all checkpoints in Section 3 with the downstream protocols described above. We make three observations based on Figure 17: (i) Downstream compute multipliers are noisier than those for validation loss, especially for CORE scores, but they are consistently larger than one for most variants, especially Untied-Grow. (ii) We fit a linear relationship between CORE NLL and validation loss and plot the residuals. Negative residuals mean better downstream performance at the same loss. We observe that Untied-Grow is consistently above Vanilla. (iii) We visualize the DCLM CORE scores for one checkpoint each from Vanilla and Untied-Grow. Despite a slightly smaller compute budget, Untied-Grow has better scores in all five categories except commonsense. These results suggest that Untied-Grow has an inductive bias favoring some downstream tasks, and they also explain why compute multipliers for downstream performance are larger than those for validation loss in Figure 3.




(a) Downstream scaling ladders.




(b) CORE NLL versus validation loss and fit residuals.




(c) DCLM CORE category scores.

Figure 17: **Downstream performance across architectures and categories. (a):** Downstream evaluations on the checkpoints shown in Figure 2. DCLM CORE evaluations are noisy, as are the compute multipliers. In CORE NLL and accuracy on filtered tasks, we see compute multipliers consistently larger than one. Untied-Grow maintains a consistent $2\times$ compute multiplier over Vanilla. **(b):** A linear fit between CORE NLL and validation loss (left) and its residuals (right), with both axes reversed. This compares downstream performance at matched loss. **(c):** The 22 tasks in DCLM CORE are grouped into five categories, and the average adjusted accuracy is reported for Untied-Grow $d16$ and Vanilla $d20$. Despite using $15\%$ less training compute, Untied-Grow outperforms Vanilla in four out of five category scores.

### C.2 Prescriptive Downstream Scaling Laws

In practice, final target runs are often planned around either a compute budget or a specific downstream goal. In this section, we follow the latter approach and design a run that is intended to match GPT-3 13B's capability (Brown et al., 2020) on DCLM CORE (Li et al., 2024; Karpathy, 2026). To predict the required compute, we need a downstream scaling law. Inspired by Grattafiori et al. (2024), we fit a taskwise pipeline: compute to answer NLL, answer NLL to centered accuracy, and task accuracy to the full CORE score. Neither validation loss nor the aggregate CORE NLL is an intermediate in this forecast.

Table 11: Tasks excluded from the fixed 17-task subset. Accuracy gaps are relative to chance at the largest-token Vanilla checkpoint, in percentage points (pp); Kendall's $\tau_b$ is measured across the eight Vanilla checkpoints.

| Excluded task | Reason for exclusion |
|---|---|
| CommonsenseQA | Low rank correlation ($\tau_b = 0.000$) |
| BoolQ | Below chance ($-4.62$ pp); low rank correlation ($\tau_b = 0.286$) |
| BIG-bench Language Identification | Low rank correlation ($\tau_b = 0.143$) |
| BIG-bench CS Algorithms | Rank correlation just below 0.50 (rounded $\tau_b = 0.500$) |
| BIG-bench Repeat Copy Logic | At chance ($+0.00$ pp); low rank correlation ($\tau_b = 0.423$) |

1. **Compute to task NLL.** For series $a$ (an architecture or corpus) and each of the 17 Vanilla-filtered tasks $t$, fit $B_{a,t}(C) = E_{a,t} + A_{a,t}(C/10^{18})^{-\alpha_{a,t}}$ with Huber loss. Each series has its own task-NLL scaling laws.
2. **Task NLL to accuracy.** Fit one sigmoid per task, $\hat{c}_t(B) = [1 + \exp(s_t B - b_t)]^{-1}$, shared across the series being compared. The target is the task's centered accuracy.
3. **Aggregation.** Average the 17 predicted task scores and fit a shared, no-intercept coefficient $r$ to recover full CORE accuracy: $\widehat{\mathrm{CORE}}_{22,a}(C) = \frac{r}{17}\sum_t \hat{c}_t(B_{a,t}(C))$.

Composing these three steps gives a downstream scaling law, as shown in the right panel of Figure 3. To reach the target CORE score, we invert the pipeline to estimate the required compute, which corresponds to a $d26$ model for Untied-Grow. As shown in Figure 3, the extrapolated $d26$ lies near the scaling law predictions.

### C.3 Transfer from FineWeb to FineWeb-Edu

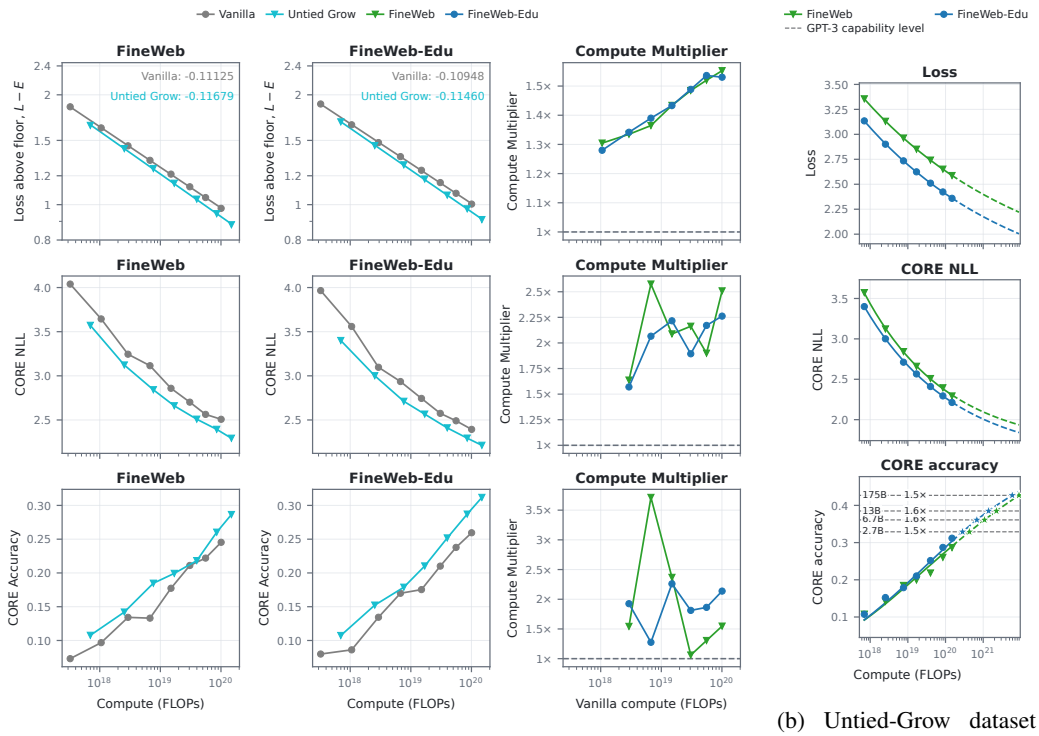
In Section 3.3, we repeat the Vanilla and Untied-Grow ladders on FineWeb-Edu with the same training recipes from FineWeb. We make two observations from Figure 18a: (i) Improvements in loss and downstream metrics for Untied-Grow over Vanilla are similar on both datasets. (ii) Switching from FineWeb to FineWeb-Edu gives a constant compute multiplier for downstream performance. This justifies our choice to switch from FineWeb to FineWeb-Edu in Section 3.3 because we want to achieve GPT-3 13B performance efficiently.

### C.4 Additional datasets

In the previous subsection, we see a constant multiplier improvement from FineWeb to FineWeb-Edu. We expand the comparison by running on additional datasets. Another motivation comes from a recent blog post by Patel & Han (2026), who train models with datasets and architectures from different years and conclude that progress in deep learning primarily comes from better datasets, while architectures have a small impact. While their architectural changes differ from the techniques we consider, the idea of going back to older datasets and looking at scaling laws is interesting.

In this section, we adopt a simplified version of the tuning protocol in Appendix A to save node hours, and we verify that it is as optimal as the original tuning protocol. For fair comparisons, all results in this section use the simplified version. We consider three datasets: OpenWebText (Gokaslan & Cohen, 2019), Pile dedup (Gao et al., 2020), and FineWeb (Penedo et al., 2024). We run Vanilla and Untied-Grow. Figure 19 compares their scaling and downstream performance across the three datasets.

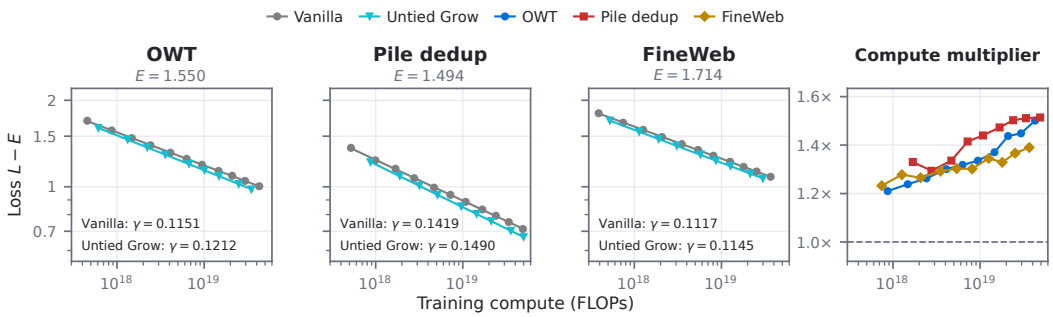
We make a few observations: (i) Untied-Grow's exponent improvements over Vanilla are dataset dependent. Pile dedup shows the largest exponent improvement of 0.007. (ii) On DCLM CORE, the scaling behaviors are similar across the three datasets, with Pile dedup having a small benefit. On OLMES, the benchmark that Patel & Han (2026) use, we see that later datasets scale significantly better than earlier ones. This shows that downstream performance scaling depends on specific tasks and that performance can vary drastically. (iii) As in Figure 17b, we plot the fit residuals for two benchmarks and three datasets, and we find that Untied-Grow consistently has better downstream performance than Vanilla when loss is matched.



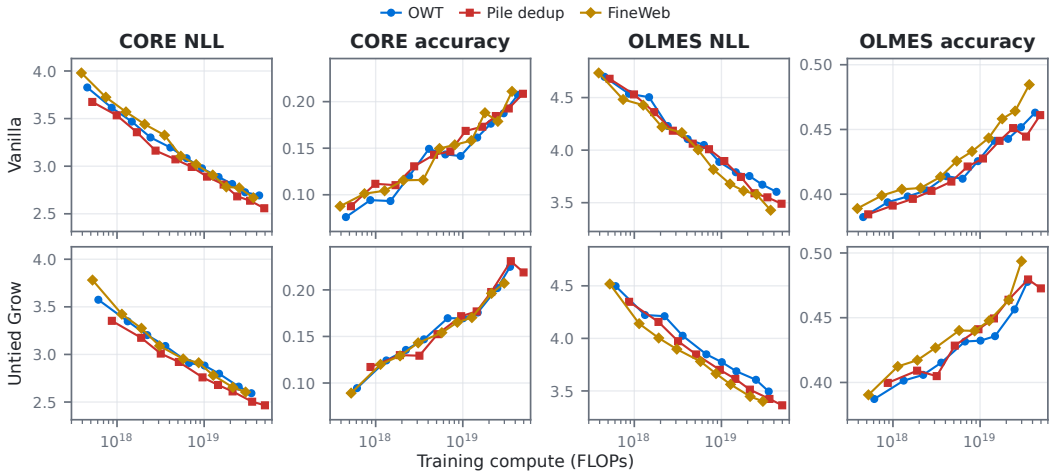

(a) Architecture gains on FineWeb and FineWeb-Edu

(b) Untied-Grow dataset comparison

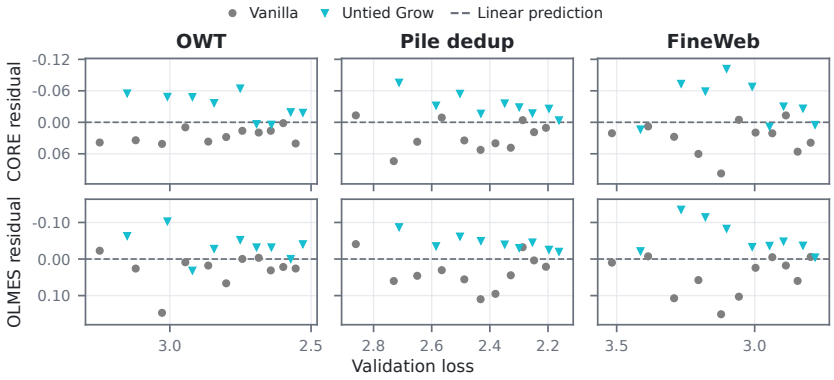
Figure 18: **Transfer from FineWeb to FineWeb-Edu. (a):** Comparison of loss, NLL, and CORE accuracy between Vanilla and Untied-Grow on FineWeb and FineWeb-Edu. The first two columns show the two datasets, and the third column shows the compute multipliers. Different rows show different performance measurements. Loss exponent improvements and compute multipliers on all metrics are similar for FineWeb and FineWeb-Edu. **(b):** Untied-Grow dataset comparison. Loss is not comparable between datasets. CORE NLL and accuracy are comparable. CORE accuracy scaling laws are modeled by the pipeline in Appendix C.2. Horizontal dashed lines mark GPT-3 CORE scores and the compute multipliers of the two datasets. We see a constant multiplier improvement in CORE accuracy when switching from FineWeb to FineWeb-Edu.




(a) Loss scaling and compute multipliers




(b) CORE and OLMES across datasets




(c) CORE and OLMES NLL residuals

Figure 19: **Scaling on additional datasets.** All 60 checkpoints use the simplified protocol: Vanilla $d6$–$d16$ and Untied-Grow $d6$–$d14$ at every integer depth. **(a)** Reducible validation loss and loss-matched compute multipliers on OWT, Pile dedup, and FineWeb, interpolated without extrapolation. **(b)** CORE and OLMES NLL and accuracy, with Vanilla above Untied-Grow. **(c)** CORE (top) and OLMES (bottom) NLL residuals from one pooled linear fit per dataset and metric. Both axes are reversed, so negative residuals appear higher.

# D Additional Results for Data-Constrained Scaling

We extend Section 4 across repetition levels, weight decay, and architecture controls. At fixed token exposure, we compare spending additional compute on stored model size or extra core passes.

## D.1 Data Repetition Changes the Preferred Recurrence

Figure 20 compares approximately 1B token exposures from fresh data, a 250M-token pool repeated four times, and a 100M-token pool repeated ten times. The first three columns use weight decay 0.8 and otherwise fixed base hyperparameters. Under fresh data and four epochs, the preferred recurrence remains near one to two passes across the measured budgets. With ten epochs, increasing model size at low recurrence eventually worsens loss, while higher recurrence postpones this upturn; the best recurrence therefore rises more strongly with compute.

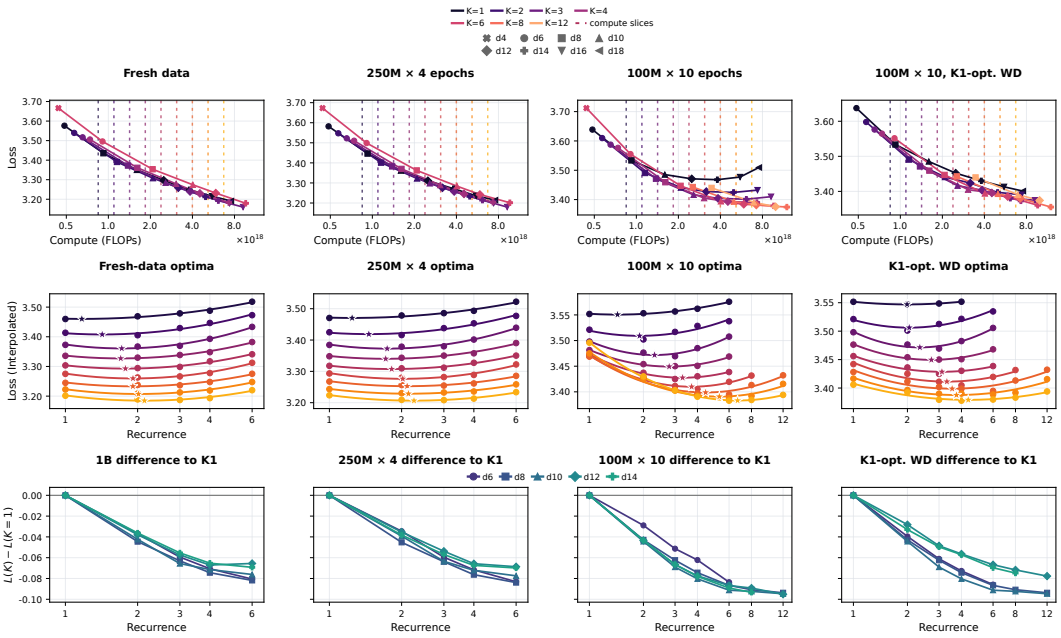



Figure 20: **Recurrence across data-repetition regimes.** Columns compare fresh data, 250M tokens repeated four times, and 100M tokens repeated ten times at fixed weight decay 0.8, followed by the ten-epoch regime with each size's $K = 1$-selected weight decay. Rows show loss versus compute, matched-compute recurrence fits, and loss differences from $K = 1$ at fixed stored depth. Colors identify recurrence in the top row, compute slices in the middle row, and depth in the bottom row. Filled stars mark interior fitted optima; hollow stars mark the best measured recurrence when the fit is censored. The ten-epoch columns include $K = 8, 12$ only where their measured compute ranges bracket the comparison budget.

The fourth column repeats the ten-epoch comparison with weight decay selected at $K = 1$ for each model size and then held fixed across recurrence. This reduces overfitting and the shift toward larger $K$, but does not remove the shift. The bottom row measures $L(K) - L(1)$ at fixed stored depth: extra passes continue to reduce loss even when increasing stored size becomes less useful. These fixed-depth comparisons spend more compute as $K$ grows; the middle row makes the equal-compute comparison.

At each compute slice, we interpolate within measured ranges and fit a quadratic in log recurrence and log loss. Filled stars mark interior minima; hollow stars mark the best measured recurrence when no interior optimum is identified. The ten-epoch sweeps include $K = 8, 12$ where measurements bracket the budget; the other regimes end at $K = 6$.

## D.2 Weight Decay and Recurrence Are Complementary

Figure 21 crosses the ten-epoch protocol with weight decay in $\{0.05, 0.2, 0.4, 0.8, 1.2, 1.6\}$. Weak regularization produces the strongest upturn in loss as stored size grows and favors high recurrence

at large budgets. Stronger weight decay reduces that upturn and moderates the preferred recurrence. Extra passes still improve loss at fixed stored depth, although the gains are smaller. Thus, recurrence gains are largest when regularization is insufficient, but are not eliminated by tuning weight decay.

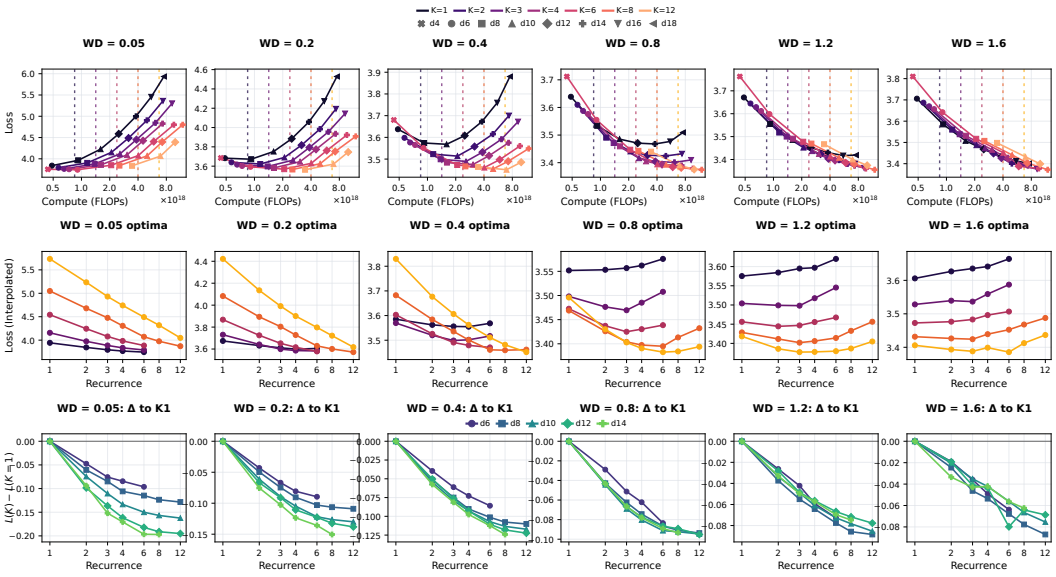



Figure 21: **Weight decay moderates the preference for recurrence.** Each column uses one of six weight-decay values under the 100M-token, ten-epoch protocol. Rows show loss versus compute, matched-compute recurrence comparisons, and $L(K) - L(1)$ at fixed stored depth. Low weight decay produces stronger overfitting with model size and a greater preference for additional passes. Compute-slice colors match between the first two rows; the bottom row uses depth colors. Comparisons interpolate within measured ranges, including $K = 8, 12$ where available. Panels use independent vertical scales.

### D.3 Hyperparameter Transfer across Recurrence

The preferred weight decay changes more with stored depth than with tied recurrence (Figure 22a). On the shared grid, selecting weight decay at $K = 1$ gives $0.4$ at $d6$, $0.8$ at $d8$–$d10$, $1.2$ at $d12$–$d14$, and $1.6$ at $d16$–$d18$. Reusing these values across $K$ yields the fourth column of Figure 20.

A separate fresh-data sweep at $d8$ tests transfer of the global learning rate, weight decay, output multiplier, residual multiplier, and injection scale across $K = 2, 4, 8$ (Figure 22b). Tied models retain broadly similar optima, while untied models shift more, particularly toward smaller learning-rate and residual multipliers at higher recurrence. Tokens are fixed within each family at approximately $1.235$B for tied models and $1.466$B for untied models. Compute increases with recurrence; for untied models, tokens per stored parameter also decrease.

### D.4 Architecture Controls: Sharing Weights and Adding Depth

Figure 23 compares three controls under the 100M-token, ten-epoch protocol at weight decay $0.8$. Tied Vanilla repeats plain Transformer cores with shared weights. Untied Vanilla uses independent copies and is equivalent to a deeper plain Transformer. Both use the frozen Vanilla recipe without the boundary operator, share the $K = 1$ ladder, and match FLOPs at each depth and recurrence. The third control varies the number of untied cores with the boundary operator.

The preferred recurrence generally increases with budget in all three families, so repetition can favor depth even without weight sharing. Both plain-Transformer controls remain above the tied boundary-operator frontier. The untied operator model is slightly better at the two smallest reference budgets but worse at the three larger ones: tied looping's advantage emerges as adding depth through new parameters becomes less effective.

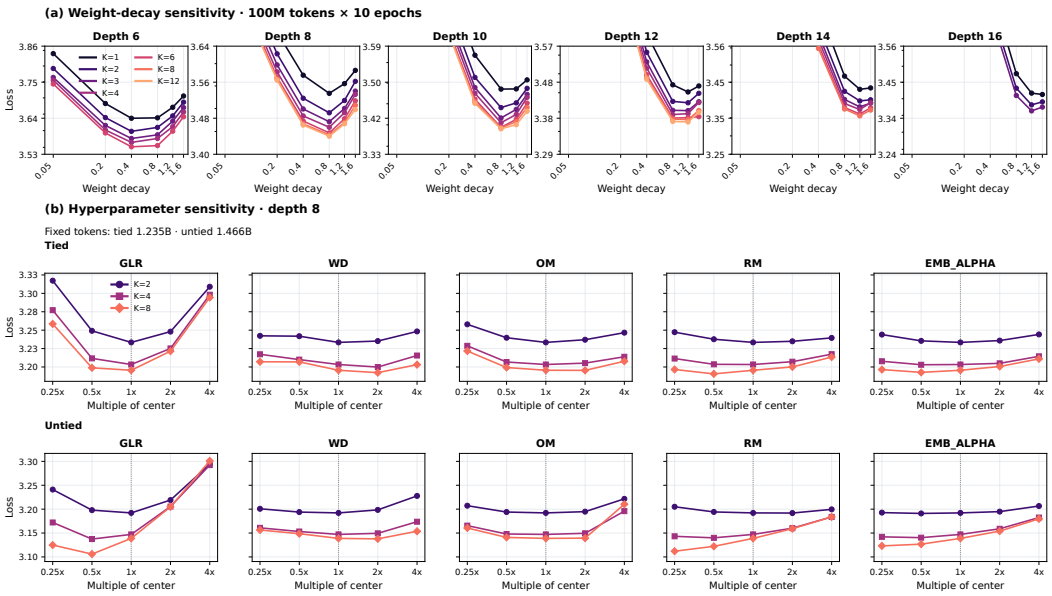



Figure 22: **Hyperparameter sensitivity to stored depth and recurrence. (a)** Weight-decay sweeps under the 100M-token, ten-epoch protocol, with one panel per stored depth. The preferred weight decay shifts with depth but varies relatively little across tied recurrence. **(b)** Fresh-data sweeps at $d8$ with two, four, and eight passes, using fixed token counts within each family: 1.235B for tied models and 1.466B for untied models. Columns vary GLR, weight decay, output multiplier, residual multiplier, and injection scale by $\{1/4, 1/2, 1, 2, 4\}$ times their base values. Dotted lines mark the base recipe. Untied recurrence causes larger shifts in several optima.

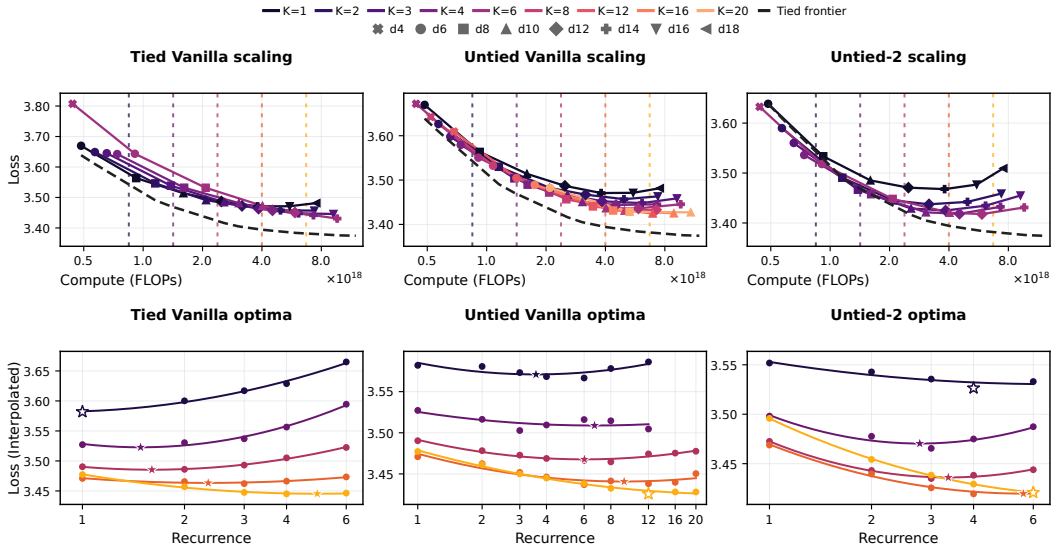



Figure 23: **Architecture controls under data repetition.** Columns show Tied Vanilla, Untied Vanilla, and the untied family with the boundary operator, all at weight decay 0.8 on 100M tokens for ten epochs. Top: loss versus compute, with the tied boundary-operator frontier repeated as a dashed reference. Bottom: matched-compute recurrence comparisons. Filled stars mark interior fitted optima; hollow stars mark censored fits at the best measured recurrence. Both plain-Transformer controls remain above the reference frontier. The untied operator family is slightly better at the two smallest reference budgets but worse at the three larger budgets.